

UC Santa Barbara

UC Santa Barbara Electronic Theses and Dissertations

Title

Programming Language Design for Timed, Concurrent Systems

Permalink

<https://escholarship.org/uc/item/80j8n51z>

ISBN

9798186382089

Author

Kringen, Harlan

Publication Date

2026-06-12

Peer reviewed|Thesis/dissertation

University of California
Santa Barbara

Programming Language Design for Timed, Concurrent Systems

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Harlan Kringen

Committee in charge:

Professor Ben Hardekopf, Chair
Professor Timothy Sherwood
Professor Jonathan Balkind
Professor Aleksandar Nanevski, IMDEA

June 2026

The Dissertation of Harlan Kringen is approved.

Professor Timothy Sherwood

Professor Jonathan Balkind

Professor Aleksandar Nanevski, IMDEA

Professor Ben Hardekopf, Committee Chair

June 2026

Programming Language Design for Timed, Concurrent Systems

Copyright © 2026

by

Harlan Kringen

Curriculum Vitæ

Harlan Kringen

Education

2026 Ph.D. in Computer Science, University of California, Santa Barbara
2015 B.A. in Linguistics and Philosophy, Boston University

Courses Taught at the University of California, Santa Barbara

Spring 2026 CCS CMPSC 130E: Logic, Topology, and Computation
Winter 2026 CCS CMPSC 1L: Programming Lab

Courses Assisted at the University of California, Santa Barbara

Fall 2025 CMPSC 9: Intermediate Python Programming
Spring 2025 CMPSC 154: Computer Architecture
Winter 2025 CMPSC 154: Computer Architecture
Fall 2024 CMPSC 8: Introduction to Computer Science
Spring 2023 CMPSC 138: Automata and Formal Languages
Spring 2020 CMPSC 160: Translation of Programming Languages
Winter 2020 CMPSC 48: Computer Science Project
Fall 2019 CMPSC 170: Operating Systems

Academic and Professional Experience

Jan 2026 - Jun 2026 CCS Computing Fellow, UCSB
Oct 2024 - Oct 2025 Visiting Researcher, IMDEA
Jan 2024 Student Research Contest, POPL'23
Jun 2020 - Aug 2020 Software Research Intern, EXOS Financial
Mar 2018 - Sep 2019 Software Applications Engineer, Righthand Robotics
Jun 2015 - Oct 2017 Research Programmer, Quantlab MA

Academic Service

Spring 2026 Artifact Evaluation Committee, PLDI'26
June 2023 Student Volunteer, PLDI'22

Peer-Reviewed Publications

Semi-Automated Translation of a Formal ISA Specification to Hardware
Harlan Kringen, Zach Sisco, Jonathan Balkind, Timothy Sherwood, Ben Hardekopf
PLARCH 2023 (position paper)

PyLSE: A Pulse-Transfer Level Language for Superconductor Electronics
Michael Christensen, George Tzimpragos, **Harlan Kringen**, Jennifer Volk,
Ben Hardekopf, Timothy Sherwood
PLDI 2022

Abstract

Programming Language Design for Timed, Concurrent Systems

by

Harlan Kringen

This thesis presents three languages for reasoning about timed, concurrent, or reactive, systems. The field of reactive systems is vast, covering technologies from distributed systems to automata, from industrial process to electronic circuits. The core characteristics include sensitivity to features such as timing, communication between components, mutual access to shared data, and intentional non-termination. Research in reactive systems is traditionally associated with model checking and other forms of post-hoc verification. While this traditional approach provides a large degree of automation, covering a wide array of verification tasks, it also has several drawbacks. These drawbacks include scalability issues, absence of equational reasoning, and a lack of modularity between models and property encodings. Additionally, the field of reactive systems is relatively understudied from the perspective of programming language design and theory.

We approach reactive systems through a programming language design lens, and derive three languages for programming and verifying reactive systems. The languages build on each other in the sense that each language addresses a deficiency in the previous. The timed, concurrent system we use as a running example is superconducting electronics (SCE), an emerging hardware technology. We focus on SCE due to its lack of preexisting models, as well as its unique information encoding. While we begin by focusing on SCE, each successive language gains some degree of abstraction beyond just hardware, to the degree that the final language operates at a high level of generality. The first language, PyLSE-TA, is a specification and verification language designed to model superconduct-

ing electronics in particular. The second language, Citrus, is a fully-fledged programming language for SCE and is adaptable to other systems with similar, pulse-based information encodings. The third language, Plata, is a programming and specification language that can express a broad class of reactive systems with verification facilities built into the language itself. These three languages are themselves investigations into various techniques and frameworks for reasoning about reactive systems. The repeated application results in a novel approach to reactive systems, informed by programming language theory, that is correct by construction and amenable to a high degree of verification.

Contents

Curriculum Vitae	iv
Abstract	vi
1 Introduction	1
2 Background	11
3 Specifying Mealy Machines with Timed Automata	30
3.1 Translation between PyLSE and PyLSE-TA	33
3.2 Circuit Verification	42
3.3 Results	46
4 Programming Machines, Coinductively	49
4.1 The Citrus Language	50
4.2 Equational Reasoning	61
4.3 Case Studies	68
5 Topology of Timed, Concurrent Systems	76
5.1 The Finitely Live Topology for Timed Automata	78
5.2 A Point-Free Topological Semantics for PLATA	82
5.3 The Refinement Calculus for PLATA	92
5.4 Lean Library Formalization	96
6 Ongoing and Future Work	105
6.1 Process Algebra and Communication Protocols	106
6.2 Automating Process Algebra	110
6.3 Discussion	113
7 Conclusion	115
Bibliography	117

Chapter 1

Introduction

Many systems can be modeled as timed events over networks of concurrent objects, such as hardware design [1, 2, 3, 4], network routing [5], and timing-sensitive secure information flow [6, 7]. Timed Automata (TA) [8] are a common formalism for representing such systems; they extend non-deterministic finite automata with a set of real-valued clocks and clock constraints as well as channel-based message passing. Given a TA model, the traditional way to reason about a system is to formulate properties in temporal logics such as CTL [9] and LTL [10] and apply standard model-checking algorithms [11]. Unfortunately, there are several problems with this approach.

State-space explosion is a well-known issue for model checking that limits scalability. As the size of the model grows, the size of the model-checkable graph grows often exponentially. This means that model checking tools quickly fail on moderately large systems. State-space explosion is an actively researched area in model checking and often requires subtle observations about the problem space to enable larger verification tasks. In the case of SCE, even moderately sized circuits succumb to state space explosion thus making it a severely limiting factor in larger analyses.

A lack of equational reasoning highlights the fact that models checking is confined to

a specific model, automaton, system. While the properties it can establish over a model can be relatively intricate, the properties are difficult to generalize over similar models, and in practice, must be done manually for every new model. Equational reasoning can be thought of in terms of generic programming, where properties are established between classes of models as opposed to every individual model. This is a lightweight form of verification that features predominantly in, for instance, process algebra and functional reactive programming.

A lack of modularity between TA model and property encoding is persistent throughout the verification process. In practice, feasible model-checking requires that the property being checked, as formulated in a temporal logic formula, must closely match the graphical structure of the TA itself—that is, formulating queries about a TA requires explicit knowledge of the nodes and edges of the discretized abstract version of the TA [12], and the form of the query may require changing the TA itself (e.g., adding states to represent possible error cases). This means that we cannot develop the TA and property in isolation from each other. Moreover, modeling a system at the level of a finite automaton is often difficult to do in an understandable way, even for small models. Another problem appears when thinking about the equivalence, or equality, of systems. One of the most accepted notions of equivalence for finite automata is bisimulation. This is a relation on nodes and edges such that if one automaton can perform an edge transition, the other automaton can as well, and constitutes the largest possible such relation. Bisimulation suffers from the aforementioned two problems as it is as classically formulated over the transition system.

A lack of modularity between multiple TA models means that the model of a system is monolithic in the sense that it represents the entire system. Due to the fine-grained nature of the model it is typically difficult to compose with other models or extend based on new information. In practice, this means that more than likely, the model must be

scrapped and redesigned from scratch.

Ultimately, these problems are largely due to the post-hoc nature of analyzing reactive systems. Programming language design and theory has alternatively produced several techniques that focus instead on writing code that is “correct by construction”. For instance, the use of type systems enforces constraints on compositionality of structures from the outset. The use of denotational and mathematical semantics allows for abstracting behavior away from purely syntactic representations. The emergence of theorem provers has enabled a flexible forms of verification, from automatic/push-button verification, to fine-grained manually proofs. This gives us the central question of this dissertation:

Thesis: *How can programming language design offer new ways to reason clearly about timed, concurrent systems?*

We answer this question by designing three, novel programming languages. We begin with a specification and verification language for SCE circuits, specifically building on top of the existing PyLSE language. This language, PyLSE-TA, translates existing specifications for SCE circuits into timed automata, schematizes correctness properties of the circuits into logical formulas, and then uses an industrial model checker to verify the circuits. This approach resides solidly in the current state-of-the-art in terms of reactive systems, namely, model checking and therefore suffers from the problems listed above. Establishing this foundation is crucial to understanding how we can build an alternative. The second language, Citrus, formalizes the specification of SCE circuits as functional components, prioritizing compositionality from the outset. It encodes the components in a coinductive type system, thereby additionally building in the inherently non-terminating aspect of the circuits. Moreover, this approach allows for a straight-

forward encoding of timed bisimulation that benefits from its own proof principle. The Citrus language is easily adaptable to other systems that primarily communicate via instantaneous, discrete “pulses”, which broadly includes neuromorphic computing, photonic computing, and distantly, distributed communication schemes. To this end, we attempt to generalize beyond Citrus. With this goal at the forefront, we construct the Plata language. Plata is structured around predicate transformers which work to both specify and implement a model. The predicates refer to properties that the design must ensure over the course of its execution, e.g. timing properties via the values of clocks, or contents of a state. Predicate transformers enjoy a refinement relation in which the transformers between them can be made more and more detailed while still provably maintaining the predicates. In this way, we use predicate transformers to not only specify the properties we desire but also incrementally refine them into valid designs.

Having introduced the three languages we set up the running example of superconducting electronics (SCE). As mentioned, SCE is a prime example a timed, concurrent system. Superconducting electronics (SCE) are an emerging technology that promises ultra power-efficient circuits. However, SCE operate on asynchronous, pulse-based logic rather than the traditional synchronous, level-based Boolean logic of CMOS circuits that are the most common implementation of modern commercial electronics. Information in a SCE circuit is transmitted via near-instantaneous pulses, with the relative timing of pulses on different wires conveying semantic meaning. SCE as a technology is under active development and there are no established formalisms for describing or reasoning about SCE circuits. Most commonly, SCE circuits are described via low-level physical circuit diagrams in a tool such as SPICE [13]. In this way, SCE are paradigmatic reactive systems, containing a large amount of timing-sensitive information and network-level concurrency.

We know of one existing work which takes a programming language perspective on

SCE, PyLSE [1]. PyLSE represents SCE as “PyLSE Machines”, Mealy-machine-like functions, keeping track of all of the necessary timing and state information internally. As a tool, it allows for rapid prototyping and simulation, as well as compilation to Verilog code. By default, it lacks formal verification tools, however. PyLSE-TA patches this problem by mapping PyLSE Machines to timed automata through a series of translation steps. These translation steps work to convert functional I/O to message passing, ensure timing constraints are maintained and automatically deduce correctness queries. PyLSE-TA therefore greatly expands on the capabilities of PyLSE, taking it from a specification and prototyping language to a verification language as well. The work in PyLSE-TA sets up much of the rest of this dissertation, by revealing the shortcomings of reactive system design and its attachment to model checking and post-hoc verification.

While PyLSE-TA is a useful tool for describing and simulating SCE circuits, it does not allow for *reasoning* about such circuits at the PyLSE Machine level of abstraction. For example, we cannot reason about the equivalence of two PyLSE automata, or whether a particular transformation on PyLSE automata (e.g., to optimize an SCE circuit) is guaranteed to preserve equivalence. The main contribution of this language is to provide a framework for algebraically reasoning about PyLSE *Machines*. That is, we look to the features of the PyLSE Machines as guides to improve upon PyLSE-TA. Our framework is inspired by the concept of *functional arrows* [14], which have been applied in fields as diverse as graphics [15] and robotics [16]; we show that they can also be applied to the domain of SCE to create a rich, expressive language for algebraically reasoning about PyLSE Machines. We formalize our framework in a language called Citrus, implemented using the dependently-typed language Agda [17], and demonstrate its usefulness via a number of case studies to prove SCE equivalencies that PyLSE and PyLSE-TA cannot establish (and, in fact, that have never been proven before).

The key insights behind Citrus are that (1) Mealy Machines, as transducers, describe

relations between sets; (2) we can describe these relations independently from Mealy Machines in a well-typed way inspired by *arrows* [14]; and (3) we can specifically use a structure akin to *looped arrows* [18] to incorporate and track the necessary state and timing information required by PyLSE Machines (which augment Mealy Machines with timing information). These time-tracking looped arrows are regular arrows augmented with an additional abstraction for tracking timing that is necessary for specifying the timing-sensitive nature of SCE. We call the Citrus data structure describing a particular combination of timing-sensitive arrows a **PulseGate**. In Citrus an SCE circuit, using the PyLSE Machine abstraction, is described in a functional, well-typed way as a composition of **PulseGates** allowing it to respect the inherently timed and stateful nature of SCE gates.

Given two Citrus programs we can equationally reason about their equivalence according to multiple types of bisimulation. We concentrate in this paper specifically on adapting *time-abstract weak bisimulation* [19]. A weak bisimulation means that the two programs are allowed to take differing numbers of steps; time-abstract similarly means that the two programs are allowed to take differing amounts of time (i.e., the circuit time delays can be different). We argue that this notion of equivalence is well-suited for reasoning about SCE circuits. Intuitively, since we want to reason about circuit equivalence to verify optimizations (which inherently change circuit timing) a strong notion of timing bisimulation that doesn't allow for different timing characteristics doesn't make sense. Time-abstract bisimulation is suitable because SCE circuits are designed so that a specific sequence of inputs will cycle the circuit through a set of states and then back to the original state. In this case, "state" doesn't refer to a specific node in an automaton, but simply whatever data is being kept track of in a feedback loop. An SCE circuit begins in some initial configuration in which it is ready to receive inputs. It then receives pulses, after which it may be unable to receive new inputs, for instance if it is delaying

or waiting for more inputs. Upon satisfying its remaining conditions, it then returns to its original configuration. Two time-abstract weakly bisimilar Citrus programs are guaranteed to have the same behavior on the same input sequences, which is enough to establish a useful notion of functional SCE circuit equivalence. We adapt this notion of equivalence to our `PulseGates` and their unique notion of cycling. These proofs are all mechanized via Agda’s dependent type system.

With the language above, we can program, in a correct by construction fashion a large class of interesting SCE circuits; however, it is by no means confined to SCE circuits alone. We can not only generalize the type system to a more expressive set of types, but we can also further build on the verification system. The Mealy machines used can be generalized to arbitrary predicate transformers and the verification technique can be expanded to incrementally refining more than just a bisimulation relation.

PLATA is a DSL and library written in the theorem prover Lean [20] and is based on a novel topological semantics of timed events and builds on the Refinement Calculus [21, 22], which provides a principled method to build PLATA programs on top of the topological semantics. PLATA allows for algebraic, equational reasoning about models, including properties of individual models and relations between models (such as equivalence).

In a topological setting, system models can be expressed using *locales*, which are lattice-like structures (see Chapter 2 for background on topological concepts). The theory of locales is the basis of the point-free approach to topology and is at the center of Stone Duality [23], which says that the open subsets (or “opens”) of a topological space form a certain kind of lattice and that considering only these opens is sufficient to treat many of the core concepts of topology, from continuity to compactness. By modeling a system using locales we can treat the model algebraically, independently from any notion of “state” or any graphical structure such as an automaton. Vickers [24] shows

that we can enrich locales with sets of points in a systematic way to create (or regain) concrete topological spaces—using this observation we can choose to enrich the locales with different sets of points (such as timed event traces) and thereby move between a variety of concrete topological spaces for interpreting and analyzing the system, thus connecting to a large theoretical toolbox.

The topological semantics gives meaning to models and enables reasoning about models, but we also need a principled way to build models. For this purpose we leverage the Refinement Calculus of Back and von Wright [21] and Morgan [22]. The Refinement Calculus provides a way to refine abstract specifications of programs into concrete representations using predicate transformers. The mathematical foundation of the Refinement Calculus can be conceived of topologically [25], where predicate transformers are homomorphisms between spaces, and this observation allows us to instantiate our own theory as an instance of the Refinement Calculus. By doing so we are able to write system specifications abstractly based on timing constraints and general behaviors, and then refine these constraints into concrete implementations that can be reasoned about.

The topological semantics plus Refinement Calculus approach that we describe in this paper is complementary to the TA plus model checking approach, not a replacement. It provides a new perspective on modeling and reasoning about timed events over networks of concurrent objects that we believe will be useful to the community. Both approaches have their advantages and disadvantages, and certain things will be easier in one setting than the other. One of the main advantages of the model checking approach is that it can provide push-button automation for checking properties, whereas the topological approach is more amenable to manual (or mechanically assisted) proof. In the topological setting, proving that a model has a certain property amounts to proving that a set of allowable timed event traces is a subset of the set of accepted (or generated) traces. However, the topological approach allows for equational reasoning that makes, for exam-

ple, equivalence properties much easier to prove. We believe that TA and PLATA models should be inter-convertible and provide a brief discussion of the issue, but leave a full exploration of this question to future work.

Finally, we also discuss in Chapter 6 ongoing work in which we formalize concurrent algorithms in the theorem prover Lean using the theory of process algebra. Process algebra is an approach to writing and verifying concurrent systems based on axioms that relate how processes can behave, for instance, how two processes could communicate in parallel. The foundation in equational axioms means that analysis can be done in pen-and-paper styles similar to mathematics or physics. Many concurrent algorithms, in the form of communication protocols, have been verified in hand-written proofs using process algebra [26, 27, 28, 29]. By formalizing a fragment of the theory of process algebra in Lean, we show how this enables a high degree of automation, turning previously manual proofs into automatable proofs. This can be seen as a supercharged version of our previous work on equational reasoning in Citrus and specification/refinement in PLATA.

In summary, the work in this dissertation is a formalization of three languages, PyLSE-TA, Citrus, and PLATA in the languages Uppaal and Python, Agda, and the Lean theorem prover [20]. Each represents a distinct contribution to the space of timed, concurrent systems and together represent a contribution to the space of correct-by-construction techniques, and more broadly, the utility of programming language design on the field of reactive systems.

Contributions. This dissertation is structured as follows:

- PyLSE-TA, a novel translation of PyLSE into timed automata, permitting push-button verification of SCE.
- Citrus, a coinductive semantics for SCE that is the first to notice the underlying lattice structure of the core operations.

- PLATA, a novel topological semantics for timed, concurrent events, including an instantiation of the Refinement Calculus allowing the use of incremental refinement programming techniques to build system models

Chapter 2

Background

As this thesis builds three different languages, each representing roughly a different framework from the field of programming language design, there is a substantial amount of background material to cover. To understand PyLSE-TA, we first canvas the preexisting PyLSE language followed by the theory of timed automata, and in particular, the variant used in the Uppaal model checker. For Citrus, we explain the core features of functional reactive programming with arrows as well as the basics of coinduction as implemented in Agda. Finally, for PLATA, we provide a basic introduction to the point-set and point-free topology, as well as the Refinement Calculus.

2.0.1 Superconducting Electronics (SCE)

Although our present work abstracts much of the hardware complexity of superconducting electronics, we provide here a brief explanation of the basic principles of SCE. In SCE, a “flux quantum” carries the unit of information. In practice, these are *pulses* which occur at very low temperatures [30]. Pulses last on the order of one picosecond, allowing for subterahertz frequencies. This technology provides for high-throughput circuits with power consumption three orders of magnitude lower than conventional semiconductor-

based circuitry (CMOS). SCE gates are conceptually similar to CMOS gates in some ways, but differ in others. Gates are functional objects which are connected by wires on which pulses arrive and depart. The main difference SCE gates have from CMOS gates is that, while the pulses themselves are instantaneous and transient, the gates are stateful. That is, the gates respond to pulses by internally changing their state. SCE gates can then keep track of when pulses arrived and reset to an initial configuration, e.g., after some time has passed or when another input has arrived.

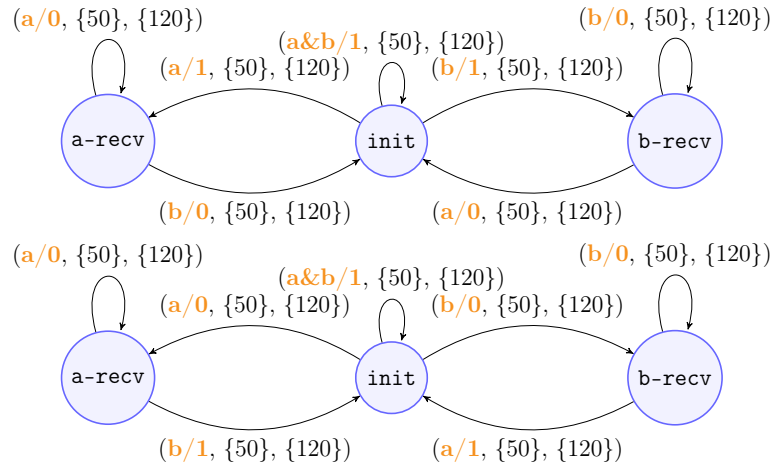


Figure 2.1: First Arrival (FA) and Last Arrival (LA) SCE gates as timed Mealy Machines, with FA on the top and LA on the bottom. The syntax “(a & b)/1” covers the case that the “a” and “b” pulses arrives simultaneously. The I/O information is colored in orange while the timing information, colored black, is given as a tuple next to it. The tuple includes the transition delay and the firing delay. Notationally, the tuple “(a/1, 50, 120)” indicates an “a” pulse arrives, triggers a delay of 50 picoseconds as the gate transitions, and then triggers a pulse as an output which results in a 120 picosecond firing delay. An “a/0” means that no pulse is triggered.

Figure 2.1 describes two SCE gates, First Arrival (analogous to a conventional OR gate) and Last Arrival (analogous to a conventional AND gate) using Mealy Machines (i.e., a finite state transducer) augmented with timing information. An FA gate pulses

when it receives the first of two possible signals and resets once it has received the second; an LA gate pulses when it receives the second of two possible signals and immediately resets.

A Mealy Machine can be represented as a function of type $(Input \times State \rightarrow Output \times State)$. The notation $x/1$ on an edge means the gate receives a pulse on the x wire (or, in the theoretical model, reads an x symbol from the input) and then sends a pulse on the output wire (writes a 1 symbol to the output); the notation $x/0$ means the gate reads a pulse on the x wire and *doesn't* send a pulse on the output wire (indicated in the theoretical model by writing a 0 symbol to the output).

The First Arrival (FA) gate begins at the *Init* state and waits to receive either an a or b pulse. If it receives an a pulse it fires, that is, sends the output pulse 1. After this event it waits for the opposite pulse, in this case the b pulse, in the *a arrived* state. While waiting, if it receives another a pulse it outputs 0, indicating that it does *not* output a pulse, and stays in the *a arrived* state. If it sees a b pulse while in the *a arrived* state it outputs a 0 (no pulse) and returns to *Init*, i.e., it resets to its beginning configuration. The case is symmetric for beginning with a b pulse. Note again that pulses are associated with wires, that is, an a pulse means a pulse received on the a wire and a b pulse means a pulse received on the b wire; pulses themselves are indistinguishable beyond what wire they are on and when they arrived.

The Last Arrival (LA) gate is largely dual. After receiving an initial pulse, it does not output a pulse and instead begins waiting for the opposite pulse. Only after receiving *both* pulses does it fire and return to the *Init* state. If we consider only the orange I/O information in Figure 2.1, we get the simple, untimed description of these gates. The additional tuples of numbers labeling the transitions indicate some of the timing information that is associated with all SCE gates. Upon receiving a pulse, for instance, the FA gate spends 50 picoseconds before updating its internal state, and there is a delay

of 120 picoseconds before the output pulse appears at the receiving gate downstream.

We will focus on two aspects of timing information common to all SCE gates, namely, transition delay and firing delay. Transition delay refers to the amount of time a gate spends before it reconfigures its internal state. For instance, a pulse may arrive at a gate, which then propagates the pulse, but then spends 50 picoseconds transitioning before returning to its original configuration. A pulse arriving during this transition delay would be ignored by the gate and possibly indicates an error in the design. If a gate propagates, or “fires”, a pulse, there is also a certain amount of time spent before the pulse appears to whatever downstream gate receives it, called the firing delay.

2.0.2 PyLSE

The PyLSE language defines two main representations for dealing with SCE: The first is the PyLSE Machine which is an extension of the basic Mealy Machine to include the variety of timing information needed to describe hardware superconductors; the second is the timed automaton derived from a given PyLSE Machine. The first representation is useful for rapid prototyping and simulation while the second representation is geared to model checking. We reproduce an example of the Last Arrival circuit in PyLSE in Figure 2.2. The states of a PyLSE Machine are abstract states, while the majority of the information resides on the edges. The information on each edge is a 3-tuple consisting of “Trigger” information, “Firing Outputs” and “Past Constraints”. Triggers and Outputs have names and durations (transition times and delays, respectively). Past Constraints define minimum times required until a new pulse may be ingressed. Designing larger circuits in the PyLSE language is done procedurally by assigning outputs of circuit elements to temporary variables which are then used as inputs to other circuit elements.

```

class C(SFQ):
    ''' C Element, Default numbers come from [1] (Table 1, LA cell).
    '''
    name = 'C'
    inputs = ['a', 'b']
    outputs = ['q']
    firing_delay = 8.0
    transitions = [
        {'id': '0', 'source': 'idle', 'trigger': 'a', 'dest': 'a_arrived',
         'priority': 0},
        {'id': '1', 'source': 'idle', 'trigger': 'b', 'dest': 'b_arrived',
         'priority': 0},
        {'id': '2', 'source': 'a_arrived', 'trigger': 'b', 'dest': 'idle',
         'firing': 'q', 'transition_time': firing_delay},
        {'id': '3', 'source': 'a_arrived', 'trigger': 'a', 'dest': 'a_arrived',
         'firing': 'q', 'transition_time': firing_delay},
        {'id': '4', 'source': 'b_arrived', 'trigger': 'a', 'dest': 'idle',
         'firing': 'q', 'transition_time': firing_delay},
        {'id': '5', 'source': 'b_arrived', 'trigger': 'b', 'dest': 'b_arrived',
         'firing': 'q', 'transition_time': firing_delay},
    ]
    jjs = 5

```

Figure 2.2: The Last Arrival gate as described in PyLSE.

2.0.3 Timed Automata

The PyLSE enables rapid construction and testing of SCE circuits but does not offer features for proving correctness. To remedy this, we seek a formal model for the underlying behavior of SCE. The PyLSE library includes several basic SCE circuits which can all be given automata descriptions as detailed above. A natural model, especially given the sensitivity to timing constraints is Timed Automata.

Timed automata (TA) extend finite automata by adding a set of real-valued clocks and a simple arithmetic language to construct expressions, called *guards*, to check the passage of time. Guards may be added to any edge in the automaton to describe when that edge may legally be taken. All clocks count up at the same rate and may be reset to zero at any time. If an edge does not have a clock guard attached then it is taken instantaneously. Multiple TA can operate in parallel and communicate through action labels in a send/receive paradigm. We use ‘!’ to indicate sending actions and ‘?’ to indicate receiving actions. TAs communicating in parallel may be statically compiled to

a single TA where nodes represent certain products of the nodes of the original TAs, thus TA are closed under parallel/message passing composition. An example of a (network of) TA is given in Figure 2.3.



Figure 2.3: A network of two timed automata where we use the $||$ symbol to indicate the two are in parallel. The TA on the left will move from location A to B by the $push!$ action only after at least 5 moments have passed. Upon doing this, it will send a "push" message. The TA on the right will wait for the "push" message, at which point it will take the transition.

There is a large body of research concerning TA, beginning with their introduction by Alur in [8], which established many of the core language-theoretic results about TA. One of the most important developments is the region graph [5], later improved to the zone graph [31], which allows efficient model checking of TA. The region graph is a finite approximation of the original automaton based on a static analysis of all the intervals in which transitions are possible. However, construction of the region graph takes time exponential in the number of clocks. The zone graph has better time complexity but further coarsens the state space. Model checking queries over TA are typically written in LTL, CTL, or modal μ -calculus .

For any TA we can associate a set of accepted traces. We first define a tuple $\langle l, \eta \rangle$. The component l is a node in the TA and the component η is a set of values for each relevant clock. A trace is a sequence of such tuples, additionally keeping track of whether a tuple moves by delay or by an action to the next tuple. For example, we could have the following accepted trace for the TA on the left in Figure 2.3:

$$\emptyset \xrightarrow{2.1} \langle q_1, x := 2.1 \rangle \xrightarrow{5.0} \langle q_1, x := 7.1 \rangle \xrightarrow{push!} \langle q_2, x := 7.1 \rangle \xrightarrow{1.9} \langle q_2, x := 9.0 \rangle \dots$$

The set of all accepted traces completely characterizes the behavior of a TA.

Timed Automata form the backbone of the first language we describe, PyLSE-TA. This language sets up a formal model of SCE and provides a way to define correctness, as well as a set of tools for mechanically showing correctness. Nevertheless, we would like to harmonize this with the original formulation of SCE in PyLSE which uses Mealy machines.

2.0.4 Functional Arrows

The automata-theoretic presentation of SCE is useful for description and simulation of SCE circuits, but not for reasoning about those circuits. An alternative presentation is to find a structure that captures the same information but is more functional and compositional. To this end, we adapt the arrow type [32] prominently used in functional reactive programming (FRP). Arrows may be thought of as generalized functions, that is, they are types which record the domain and codomain at the type level.

Functions naturally constitute an instance of arrows, but so do other structures such as stream transformers, Moore Machines, Mealy Machines, and many others. This unified type signature allows us to define combinators, or higher order functions, which describe various kinds of compositions including sequential and parallel composition, branching choice, and even value-level recursion. In this way, arrows may be thought of as a language for dataflow graphs. These features can be bundled together into a common typeclass such as one might find in Haskell. However, arrows must obey certain laws for any types they are applied to, such as preserving identities, associativity, and others; these laws allow us to reason about arrow compositions algebraically.

We will draw inspiration from the Arrow typeclass which consists of four combinators, which are shown graphically in Figure 2.4 [33]. The lifting combinator $arr\ f$ takes a

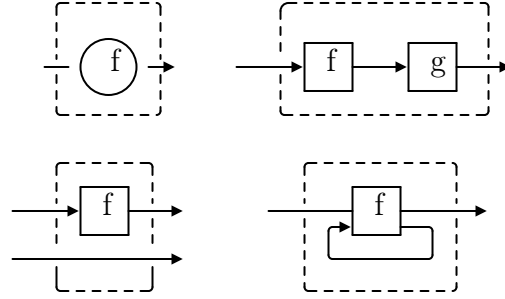


Figure 2.4: The Arrow typeclass contains several combinators which allow for the construction of complex “dataflow graph”-like terms. We implement these combinators in Citrus.

plain function and lifts it into the Arrow type. The sequence combinator $>>>$ passes the output of one arrow to the next. The *first* combinator takes two inputs but runs the arrow only on the first input. From these components we can build a parallel combinator, $***$, which runs one arrow on the first input and one on the second input, and many others. The Arrow type essentially describes stateless, feed-forward dataflow graphs.

However, to use arrows to describe functions that correspond to Mealy Machines, i.e., that have signatures such as $(Input \times State \rightarrow Output \times State)$ which persist and repeatedly respond to inputs, the basic arrow type isn’t quite enough. Instead we must take a fixpoint of the type to represent all functions defined with this signature. We use *State* as a recursion variable and rewrite the signature as $F(State) = Input \rightarrow (Output \times F(State))$. This signature denotes a function that, upon receiving an input, responds with an output and additionally a copy of itself stepped forward “in time”. These functions have no termination condition—they keep running forever, and so we use a greatest fixpoint rather than the least fixpoint represented by typical inductive types. Types defined in this way are not *recursive* but instead *corecursive*.

With the theoretical backdrop of FRP established, we look to ways for mechanically reasoning about programs built in this way. In the same way that TA added the toolbox of model checking and logical specification on SCE, we would like a similar tool to enable

reasoning about functional reactive programs.

2.0.5 Agda and Coinduction

Agda is a dependently type programming language [17]. Used as a proof assistant (i.e., an interactive theorem prover), it sits in a space similar to Rocq [34]—both languages use their dependent type systems as a way to express and prove logical properties. For this work, we concentrate on using dependent types as a way to formalize a model for superconducting circuits. The expressive power of dependent types allows us provide a DSL that can reason about complex forms of equality over circuits. Agda provides convenient facilities for working with coinductive types, as discussed below. Rocq, in contrast, makes coinductive types somewhat more awkward to work with but has greater support for proof automation (e.g., the use of tactics). Either language could be used for Citrus, but we chose to start with Agda because coinductive types are a large part of what we need to reason about SCE. It would be interesting future work to port Citrus to Rocq and compare and contrast the facilities of the two languages.

An attractive feature of Agda is its support for coinductive types. Briefly, coinductive types are a non-terminating dual to inductive types. Inductive types permit functions defined by structural recursion; these functions must be provably terminating. Dual to this, we may want to describe objects that *do not* terminate but nevertheless have enough structure to be well-behaved, such as the stream transformers described earlier. There is no end to the sequence of data and the only thing we can do with the stream is observe the most recent value. Any function that consumes this stream will be non-terminating—but as long as it only observes the most recent value, i.e., only makes use of whatever information the type provides, then the function is well-behaved in a certain way. Coinductive types work well for capturing forms of patterned repetition

in behavior. Agda internally uses a productivity checker to ensure that all coinductive values are “productive”, namely, that observations made corecursively are guaranteed to be computable in finite time [35].

In Agda, similar to Haskell, the `data` keyword establishes a data type as the least fixpoint of some type-level function. In Haskell the least and greatest fixpoints coincide, and thus `data` declarations suffice for both finite and infinite structures; in contrast for Agda we build coinductive types explicitly using the `record` construct with the keyword `coinductive`. Records may be thought of as generalized products, or tuples, defined by fields. These fields may be used to project out the data contained within the record. The intuition is that, similar to the stream example from above, the defining characteristic of records, or product types, is that we may *destruct* them to observe their values, in contrast to `data` or sum types, which focus on *constructing* terms of the type.

The infrastructure by which Agda’s typechecker is able to confirm a coinductive function is productive is based on structures called “sized types” [36]. For the programmer, sized types act like annotations to type signatures which help Agda determine *how deeply* a coinductive function is being observed or deconstructed. It is not necessary for the reader to understand these annotations, but we point them out so as to prevent any confusion in the code instances shown below. There are two main instances of sized types to be aware of. First, we use the ∞ symbol to mark a coinductive type that is “fully defined”. That is, when we want to express the equivalence of two circuits, we would like this to hold for circuits defined at all possible depths of observation. Second, we will use $\{i\}$ to denote arbitrary depths of observation and $\{j : \text{Size} < i\}$ to denote arbitrary depths below a given depth.

The second language we present, Citrus, formalizes Timed Mealy machines as arrows in the FRP sense, and combines this with a coinductive calculus to reason about reactive behavior. Thus far, we have formalized two different approaches to timed, concurrent

systems, with the running example of SCE as the main motivation. Nevertheless, it is our experiences with these two approaches, model checking and automata on the one hand and FRP and coinductive types on the other that suggest we can generalize this type of reasoning. To this end, we move to the field of topology.

2.0.6 Topology

Topological Spaces. A topological space $\mathcal{T}$ is a pair $\mathcal{T} = \langle X, \mathcal{O} \rangle$, where X is a set and $\mathcal{O}$ is a collection of subsets of X called the “observables”, “open subsets”, or “opens” [37]. The space $\mathcal{T}$ must adhere to the three rules given below. These rules ensure that (1) the carrier set X itself is open; (2) for any two subsets of X , if they are both open, then their intersection is open (note in particular that we are only considering finite numbers of intersections); and (3) infinite unions of open sets are also open.

1. $X \in \mathcal{O}$
2. $\forall s, t \subseteq X, s \in \mathcal{O} \rightarrow t \in \mathcal{O} \rightarrow (s \cap t) \in \mathcal{O}$
3. $\forall S \subseteq \mathcal{P}(X), (\forall t \subseteq X, t \in S \rightarrow t \in \mathcal{O}) \rightarrow \bigcup S \in \mathcal{O}$

Conceptually, it can be helpful to think of topological spaces as descriptions of semi-decidable properties [38]. Each open subset $o \in \mathcal{O}$ represents a semi-decidable property P_o on the underlying set X which, if it is true, can be determined true in finite time, but which cannot be proven false in finite time. For any two properties P and Q , we can imagine testing them in sequence. The result is either true if they both are, in finite time, or unknown. Clearly, this notion of sequencing only works on a finite number of semi-decidable properties at once. This idea is analogous to rule (2), interpreting intersection as sequencing. We could also test a set of properties $P, Q, R, \dots$ in parallel, so that if at

least one is true then we will discover this fact in finite time. This idea is analogous to rule (3) interpreting union as parallel composition, e.g., we allow arbitrary/infinite union.

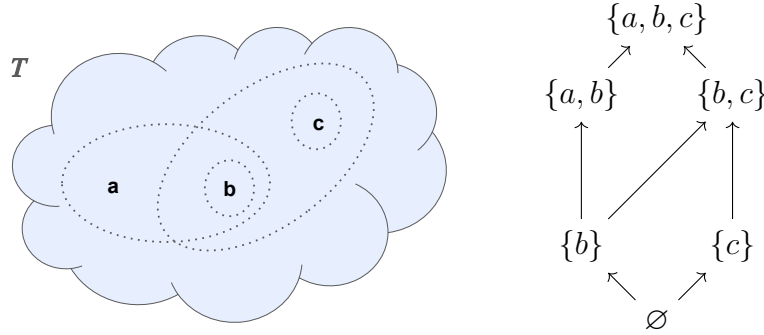


Figure 2.5: In the leftmost picture, we illustrate a topology $\mathcal{T}$ on the points of the set $X = \{a, b, c\}$ as the collection of open subsets: $o_1 = \{b\}$, $o_2 = \{c\}$, $o_4 = \{a, b\}$, $o_4 = \{b, c\}$, $o_5 = \emptyset$, and $o_6 = \{a, b, c\}$. In the rightmost image we can see that the open subsets of any space $\mathcal{T}$ are ordered by subset inclusion and form a lattice.

We can visualize a topological space as shown in in Figure 2.5 in the leftmost and center images. The first image shows a collection of points grouped into subsets which are open according to the topological definition. The second image shows that these open subsets form a lattice ordered by subset inclusion. More specifically, the lattice of open subsets is a complete lattice in which finite intersections distribute over infinite unions, i.e., it is an instance of a complete distributive lattice. Complete distributive lattices are variously called “locales”, “frames”, and “complete Heyting algebras”, among others, depending on the setting they are being discussed. We will use “locale” in the rest of the paper. Topological spaces are from the traditional “pointful” approach to topology, which defines concepts primarily in terms of the elements of the carrier set X , called “points”. In this approach the points are specified explicitly and open subsets (the locale’s elements) are defined as sets of points, with the locale’s meet and join operators being set intersection and union.

Point-Free Topology. The alternative “point-free” approach to topology abstracts away from explicit points and topological spaces, defining concepts primarily in terms of the locale structure without regard to what the “points” might be. A locale in point-free topology is any complete distributive lattice (whose elements do not need to be sets), and there is no carrier set of points. The rightmost image in Figure 2.5 shows an example of a point-free locale, which focuses on thinking not in terms of explicit open sets but in terms of “formal” opens. To show this, we replace the sets with arbitrary symbols and denote their relationships according to the meet and join of the lattice, which will replace set intersection and union.

A point-free topology may be seen as an abstraction of a pointful topological space, and a given point-free locale can be enriched with a set of concrete points to recover a specific topological space, giving a correspondence between the pointful and point-free approaches. That is, we can construct a binary relation between a locale and an arbitrary set of elements, respecting intersections and unions, in a specific way that allows us to “enrich” an abstract locale with points to obtain a concrete topological space [24].

Topologizing Labeled Transition Systems. Previous work by Clarkson and Schneider [39] has shown how to give a pointful topological semantics to labeled transition systems (LTS) via their trace sets, where points are LTS traces. We describe the construction here, and in Section 5.1 we show how we can extend the construction to define a pointful topological semantics for TA. However, we also describe the shortcomings of that approach and propose a different, point-free topological semantics in Section 5.2 that does not have those shortcomings, which is the approach we use for PLATA.

An LTS is a description of a finite or countably infinite state automaton. To show how an LTS may be given a topology, we proceed in two steps: first we derive a set of traces that characterize an LTS; then we show how this set of traces can be given

a topological structure. An LTS trace is a function from a countable index domain to values, where the values are tuples of information about the nodes and edges of the LTS. To retrieve a specific value of a trace at a particular index we apply the function to the index. We write the value of some trace t at index n as $t\ n$, using parentheses wherever it aids readability. Let L be a finite set of primitive symbols that would decorate the edges of an LTS (i.e., edge labels) and let the state space $\Sigma = S \times L$ be the product of the states and labels. Then we can define Ψ to be the set of all possible traces, where Σ^* is the set of all finite sequences and Σ^ω is the set of all countably infinite sequences over Σ .

Definition 1 (All Possible Traces)

$$\Psi = \{t \mid t : \mathbb{N} \rightarrow \Sigma^* \cup \Sigma^\omega\}$$

Let S be a set of LTS nodes, L be a set of primitive symbols (the edge labels), and $E \subseteq S \times L \times S$ represent the edges of the LTS. Let the state space Σ be $S \times L$ and let functions *state* and *label* be the first and second projections on Σ . The tuple $\langle S, L, E \rangle$ describes an LTS. The valid traces of an LTS are given in definition 2. We deal with finite traces by considering their infinite extensions, i.e., we extend a finite sequence to an infinite trace by repeating the last element.

Definition 2 (Traces of an LTS)

$$VT = \{t : \mathbb{N} \rightarrow \Sigma \mid \forall i \in \mathbb{N}, (state(t\ i), label(t\ i), state(t\ (i+1))) \in E\}$$

Now that we have characterized an LTS as a set of valid traces, we must consider which subsets of traces can form a topology. As described above, a topology is given by identifying certain subsets of a state-space as “observable” (aka “open”, etc). These subsets then need to obey the three axioms of a topological space. A subset of LTS traces $O \subseteq \Psi$ is observable if the following holds:

Definition 3 (Observables)

$$\forall (t \in \Psi_{inf}), t \in O \rightarrow (\exists (m \in \Psi_{fin}), m \leq t \wedge (\forall (t' \in \Psi), m \leq t' \rightarrow t' \in O))$$

This definition states that a set of traces all of which share (arbitrarily large) finite prefixes are open. The definition does not state that all traces in the set O are each others' prefixes, only that for any given trace, its finite prefixes are also in the set.

Proposition 2.0.6.1 (Finitely Live Topology)

The observable subsets form a topology on Ψ .

Proof: We can see that the set of all traces, as well as the empty set, are both observable. Additionally, the intersection of any two observable sets is observable as any traces remaining must have already been observable. Finally, an arbitrary union of observables must also be observable since we cannot change the status of a set of observables by adding traces (provided we also add their finite prefixes). ■

We note that the traces generated by an LTS need not be observable, or rather, they need not be open sets in the Finitely Live topology. They *are* sets that live inside the topology, however they may be open, closed, both, or neither. Clarkson and Schneider term this construction the “Plotkin” topology; however, to avoid confusion with a structure we use later on we will prefer the name “Finitely Live”, or just “trace topology”, written as $\mathcal{L}^{fin}$. In summary, we have a topological space $\mathcal{L}^{fin}$ given by $\langle \Psi, \text{Observables} \rangle$, or, the set of all possible traces as the carrier set, with the observable subsets as opens.

The original formulation of this topology can be traced to Plotkin, cited in private communication to Alpern [40]. As Clarkson and Schneider note, the definition can be seen as a reformulation of the Scott topology on ω sequences. This observation establishes a connection to traditional model checking techniques using recursion and iteration. That is, to prove a property holds of an LTS, one can iterate over trace sets, for instance, to search for a specific trace counterexample; however, the trace set or counterexample could be computationally too large or unmanageable to actually use in practice. We observe that there has been very little work capitalizing on the topological structure of

traces; we see this area as a space ripe for exploration in its own right, even beyond our use in this paper.

Connection to Safety and Liveness. We can connect the Finitely Live trace topology to the classic notions of safety and liveness. In program analysis, concurrency, and related fields, we often want to verify properties that fall into two categories, the so-called “safety” and “liveness” properties. A liveness property describes a set of behaviors in which eventually some good thing happens, such as termination. A safety property describes the case where nothing bad ever happens, for instance, the server shutting down. If we think of these properties as propositions about sequences, we can view them as trace properties, or collections of traces. In this way, we can see safety and liveness as topological properties. Alpern gives the topological definitions of safety and liveness as follows [40]:

Definition 4 (Safety)

A set $X \subseteq \Psi$ is safe if it is closed in $\mathcal{L}^{fin}$.

Definition 5 (Liveness)

A set $X \subseteq \Psi$ is live if it is dense in $\mathcal{L}^{fin}$.

A closed subset is one whose complement is open while a dense subset is one whose closure is the entire space. This connection shows an example of how we can relate topological concepts to computational concepts.

Topology, in both its point-free and point-set formulations, will be the underlying theory for our third language PLATA. And again, in the same way that Timed Automata brought with it model checking for mechanical reasoning, and we packaged FRP up with coinductive types, here we would like to combine our theoretical framework with a calculus of some kind to enable mechanical reasoning. Specifically, we would like to move into the space of “correct-by-construction” reasoning, away from the post-hoc reasoning

of model checking.

2.0.7 Refinement Calculus

The Refinement Calculus is a language constructed to use Dijkstra’s weakest precondition calculus to incrementally synthesize code from logical specifications. It was developed by both Back and Von Wright [21] and Morgan [22]. The language provides two main contributions: a way to write specifications as combinations of preconditions and postconditions, and a formalization of when one specification refines another. The language then defines several operators and lemmas that allow for incrementally, in step-wise fashion, generating more and more refined specifications until an unrefinable specification is attained. This final, unrefinable version is a concrete program that is correct by construction.

Definition 6 (Refinement Calculus Syntax)

$$D \in \text{Domain} \quad f : D_1 \rightarrow D_2 \quad \Phi : D \rightarrow \text{Prop} \quad S ::= \langle f \rangle \mid \{\Phi\} \mid [\Phi] \mid S_1; S_2$$

Definition 6 shows the formal syntax of the Refinement Calculus. Its semantics is given in terms of predicate transformers, namely the weakest precondition semantics of [41]. Predicates Φ are considered functions from some state space D to a universe of propositions (**Prop**), which we will represent in our treatment as sets **Set** D (i.e., those subsets of D for which the predicate holds). The results of the operators $\langle \cdot \rangle$ (“update”), $\{\cdot\}$ (“assertion”), $[\cdot]$ (“assumption”), and $;$ (“sequence”) are predicate transformers, i.e., functions from predicates to predicates. The result of $\langle f \rangle$ where $f : D_1 \rightarrow D_2$ is a transformer that takes a predicate Φ on D_2 and returns a predicate on D_1 by computing the preimage of Φ . The result of $\{\Phi\}$ is a transformer that, given an argument Φ' , enforces that Φ' must initially hold by intersecting it with Φ , representing a precondition. The

result of $[\Phi]$ is a transformer that, given an argument Φ' , ensures that Φ' must finally hold by unioning it with Φ^c , representing a postcondition. The sequence operator allows predicate transformers to be composed, i.e., first doing S_1 and then S_2 .

This calculus expresses a *specification* as a precondition composed with a postcondition, as in $\{\Phi_1\} ; [\Phi_2]$. The predicates Φ_1 and Φ_2 reflect propositions about the domain in question, for instance “stop light turns green in 4 seconds”, or “error rate drops below 0.5”, etc. Given domain-relevant functions f , g , and h , and a specification $\{\Phi_1\} ; [\Phi_2]$, we can, for example, successively refine the specification first to $\{\Phi_3\} ; \langle f \rangle ; [\Phi_4]$, then $\{\Phi_5\} ; \langle f \circ g \rangle ; [\Phi_6]$, and finally $\langle f \circ g \circ h \rangle$, the final concrete program.

A predicate transformer $P : \mathbf{Set} \ D_1 \rightarrow \mathbf{Set} \ D_2$ is refined by a predicate transformer $Q : \mathbf{Set} \ D_1 \rightarrow \mathbf{Set} \ D_2$ (or: Q refines P), written $P \sqsubseteq Q$, iff $\forall x : \mathbf{Set} \ D_1, P \ x \subseteq Q \ x$. A predicate transformer $P : \mathbf{Set} \ D_1 \rightarrow \mathbf{Set} \ D_2$ is monotonic iff $\forall (x \ x' : \mathbf{Set} \ D_1), x \subseteq x' \implies P \ x \subseteq P \ x'$. Refinement induces a complete lattice on monotone predicate transformers (proved in PLATA). Because locales extend complete lattices, we will see that our semantics can take advantage quite directly of the Refinement Calculus. More specifically, for PLATA our domain will be the locale of our point-free topology (which by definition is a complete lattice). This construction allows us to then apply the operators and lemmas of the Refinement Calculus to build system models by starting with an abstract logical specification of timed events and successively refining it into a concrete program using our space of predicate transformers.

With these technologies in place, we have set up the necessary background material to describe our three languages and their associated mechanizations. We can visuazlize this in the table below.

<i>Language</i>	<i>Theoretical Basis</i>	<i>Mechanical Reasoning</i>	<i>Implementation</i>
PYLSE-TA	Timed Automata	Model Checking	Python, Uppaal
CITRUS	Functional Reactive Arrows	Coinduction	Agda
PLATA	Point-free Topology	Refinement Calculus	Lean

Table 2.1: The three languages we present in this thesis can be broadly summarized according to what framework in the field of programming language design they build on, what this enables in terms of formal reasoning, and in what ambient language they were implemented in.

Chapter 3

Specifying Mealy Machines with Timed Automata

Our first language is PyLSE-TA which builds upon the DSL PyLSE [1]. Briefly, the PyLSE language is a Python DSL which models superconducting circuits as PyLSE Machines, a function-like abstraction resembling a Mealy machine. PyLSE-TA takes this specification of SCE circuits and (1) translates them into timed automata (2) automatically deduces and compiles correctness queries in the model checking logic, timed computation tree logic (TCTL). While PyLSE enables rapid prototyping and testing of SCE, PyLSE-TA takes this one step further by providing a formal foundation for the language which additionally establishes verification capabilities. Many of the timing properties of PyLSE Machines are described naturally by TA, however the fundamental difficulty is modeling the input/output behavior of the PyLSE machine. This is because PyLSE Machines represent I/O as simple functions while TA do not have a built-in notion of I/O. Instead, TA are outfitted with message passing communication over channels using send/receive-style primitives. PyLSE-TA applies a series of transformation steps to model SCE I/O that is timing-correct. With the corresponding timed automaton, we

then proceed to investigate correctness properties of SCE elements. We define properties in TCTL that express, for any SCE circuit, what it means to both operate correctly and fail explicitly, and then demonstrate how they can be deduced automatically for a given circuit. Finally, we showcase the results of model checking these correctness properties for a library of basic SCE elements. This establishes useful metrics, including the number of states required to model check a formula, the time needed, and a rough bound on the size of a timed automaton that can still be model checked successfully. The majority of this work appears directly in the PyLSE paper [1].

3.0.1 PyLSE machine

We give a brief description that summarizes the important parts. These can be seen in 3.2. The PyLSE Machine can be thought of as a functional block, taking inputs of pulses and outputting pulses. SCE differs from CMOS in that it inherently “remembers” state, as opposed to requiring a constant voltage to make information persist. To this end, the PyLSE Machine models state internally as a finite state machine. Additionally, the output pulses are gated by delays so that they may only fire after certain lengths of time. There are further complications concerning repeated pulses within certain time windows before and after a pulse is received. For instance, a repeat pulse arriving within a time window will be ignored. These properties are referred to as “setup” and “hold” time, while the time it takes to internally update state is “transition”, and finally, “firing” time refers to the length of time that must transpire before the output pulse is visible. PyLSE Machines may be composed to form networks of PyLSE Machines. These are simulated according to a dataflow style. Each step of a simulation pauses, each block is updated, followed by its successor connections are updated. Cells that must be updated are kept in a worklist. The simulation ensures that violations of timing constraints, or error states,

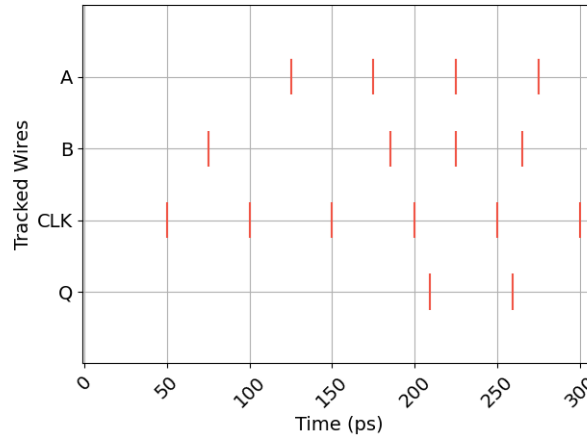


Figure 3.1: This waveform diagram visualizes when specific pulses appear on a PyLSE Machine. The pulses can be on input wires or output wires, or designated *CLK* wires.

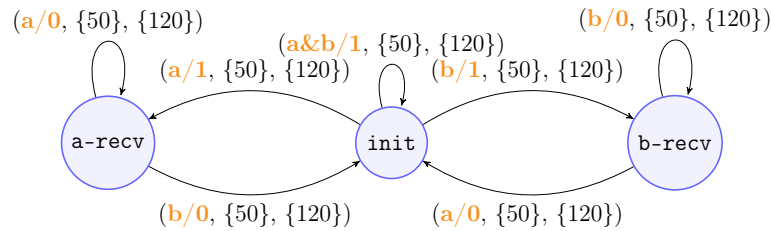


Figure 3.2: A PyLSE Machine reproduced from the PyLSE paper. This is a schematic representation of the data required to specify the data structure used in the PyLSE language. We can see inputs, *A* and *B* as well outputs, *Q*.

are propagated and flagged to the user. The internal state, as well as all of the timing constraints, are generally different for each SCE cell but act as known constants to the circuit designer and thus can be set up once and for all for each element. The interplay of these features is easiest to see in simulation traces, where given input sequences are sent into an SCE circuit and the output sequences recorded. This is visualized in 3.1.

3.1 Translation between PyLSE and PyLSE-TA

The translation from PyLSE Machine to a timed automaton is guided by the following problems. (1) *input/output* PyLSE Machines are stateful functions and so have a built-in notion of producing output given an input. On the other hand automata do not have the same notion of I/O; instead, timed automata use send/receive signals over channels. Further, while the PyLSE machines are functions, the corresponding automata must make explicit the inputs and outputs, in some sense, hard-coding them. We create two automata to bracket the SCE element itself, one to send inputs into the element, and one to accept all expected outputs. (2) *atomic actions and exact durations* We must enforce that certain sequences of actions cannot be interleaved and moreover must take an exact amount of time. (3) *error propagation* The fact that SCE are sensitive to timing conditions manifests as a wide array of error scenarios.

In the sections below, we use pseudocode to describe the translation steps. We make use of functions such as `create-node` and `.lookup` as shorthand for functions which traverse the PyLSE Machine structure. In addition to these functions, we will further use a function, `create-edge(src,chan,guard,reset-clks,tgt)` to indicate the create of an edge between two nodes, `src` and `tgt` with the addition of a `chan` on which messages are sent or received, the set of clocks to reset to zero, and finally `guard` to indicate a timing constraint based on some clock.

3.1.1 System Level Input/Output

As an SCE element is, at its core, a function, and so it can be given inputs arbitrarily. Transforming this structure into an automaton creates a difficulty in that inputs that are “sent into” the system must be explicitly encoded at the level of states and transitions. To this end, we build a “source” automaton is constructed that is able to send inputs

on appropriate channels into the circuit system itself and a “sink” automaton to receive signals from the circuit system, e.g. the outputs. There are two schemes for sending data into the system: (1) we create a source automaton with a single node but with self-edges for all possible input signals and (2) a source automaton with a node for every input signal daisy chained so that all inputs must be given according to a specific time schedule. In practice PyLSE-TA uses the second, and moreover, this is necessary for verification, as we will see in Section 3.2; however, we will describe in Figure 3.3 the first version for ease of presentation as the second is simply a more involved version of the first. We place no restrictions on the sink automaton, as it simply needs to receive outputs; if the system has been set up correctly, we can verify correctness in terms of the system and leave out any complexity on the sink automaton.

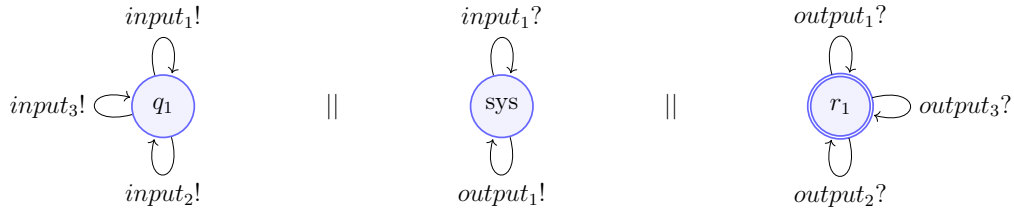


Figure 3.3: Two automata are separated by the parallel product operator ($||$). The left figure is the source automaton which sends inputs into the system while the right figure is the sink automaton that receives outputs from the system. The middle figure represents the system itself, some SCE circuit.

In Algorithm 1 we instantiate a single node, **src**, that will act as the source automaton. we scan all of the nodes in the PyLSE Machine circuit to find those that are tagged as inputs. We then create a channel for each and then for each such channel, we create an edge from **src** to **src** where the edge is labeled with the send action, e.g. the channel name and the special symbol “!”. We use the $\emptyset$ in the **create-edge** function to indicate

Procedure 1 : create-source-auto Create a source automaton from a PyLSE Machine.

Input: `circ` : PyLSE Machine

Output: PyLSE-TA

`src` $\leftarrow$ `create-node()`

`inputs` $\leftarrow \emptyset$

▷ get all of the input signals from the PyLSE Machine

for `n` $\in$ `circ.nodes` **do**

for `i` $\in$ `n.inputs` **do**

`inputs.append(i)`

▷ create a channel for each input signal, link it to the src node

for `i` $\in$ `inputs` **do**

`input-chan` $\leftarrow$ `create-chan(i)`

`create-edge(src, {input-chan}!,  $\emptyset$ , emptyset, src)`

return `src`

we are not resetting any clocks nor adding any timing constraints. The creation of a sink automaton follows a nearly identical procedure with the main difference that edges are decorated with the special symbol “?” indicating that the channel is waiting to receive data.

3.1.2 Atomic Actions, and Exact Durations

An SCE element waits for pulses on certain wires designated “input” and sends pulses on certain wires designated “output”. These are used to connect elements into larger designs. TA do not deal with I/O in a familiar, functional manner, instead splitting it up into **send** and **receive** actions. With this in mind, we similarly model I/O in two phases. We first discuss how to treat time-sensitive input, depicted in 3.5. Wires become channels, input pulses are **receive** actions modeled as `chan?`. The important distinction

from the function picture is that in the case of parallel automata, actions from other automata may interleave between the **receive** and **send** actions.

By default, the semantics of a timed automaton allow for a wide array of interleavings and possible traces. While this is a benefit of the system in the design of SCE electronics, there are certain situations in which we must finely guide the execution of actions. Conceptually, this is equivalent to creating critical regions or atomic actions. We structure our nodes and edges so that a send/receive is enforced to happen, not just enabled, as well as guaranteeing that the sequences take an exact amount of time, as opposed to simply “up to” a delay. Action transitions, such as send/receive, in timed automata are definitionally atomic, in that they do not take time to execute. Nevertheless, several interleavings can trigger this in concert with other transitions that do take time.

In the theory of timed automata implemented by Uppaal, if we want to confirm that a sequence of transitions occurs, we make use of the notion of a **committed** node. This is an extension to the base theory of timed automata. This node indicates that a trace occupying this state must immediately take the subsequent transition, therefore preventing other transitions to occur and interleave with the sequence. Further, if we want to enforce an *exact* amount of time passes during this sequence, we use a combination of timing constraints, the invariant and the guard. We create a node with an *invariant* that enforces a valid trace may occupy this state only up to the amount of time specified, e.g. may not leave later. We then *guard* the outgoing transition to enforce that the exact amount of time has passed (determined by a clock). To create the above-described critical region, we design the following gadget, shown in 3.4, where the **committed** node q2 is represented as a diamond.

In Algorithm 2 we take as input an edge in the original PyLSE Machine and expand it into a new edge with an intermediate node. This intermediate node is **committed**, which disallows intervening parallel actions. Moreover, the incoming and outgoing edges

Procedure 2 : expand-trans Expand default edge transitions.

Input: $ta : \text{PyLSE-TA}$

Input: $\text{edge} \in ta$

Output: PyLSE-TA

$t_{\text{begin}} \leftarrow \text{create-node}()$

$t_{\text{end}} \leftarrow \text{create-node}()$

▷ create the new, intermediate node which is flagged as committed

$\text{interm} \leftarrow \text{create-node}(\text{committed}=\text{true})$

▷ plumb the new states together according to the timing constraints

$\text{create-edge}(t_{\text{begin}}, \text{edge.in-channel?}, \text{clk} \geq \text{edge.delay}_1, \text{clk}, \text{interm})$

$\text{create-edge}(\text{interm}, \emptyset, \text{clk} = \text{edge.delay}_2, \text{clk}, t_{\text{end}})$

▷ we treat t_{begin} as a sort of pointer to the newly built structure

$ta' \leftarrow \text{cast-to-ta}(t_{\text{begin}})$

return ta'

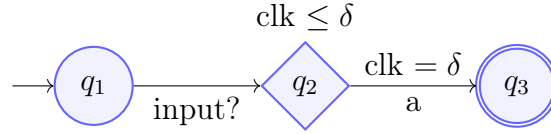


Figure 3.4: This depicts a critical region, or a sequence of actions which must occur uninterrupted by actions performed elsewhere in parallel. Here, we expect an input action, followed by an output directly after without intervening actions. We further expect the time separating these to be exactly equal to delay δ before we can take the a action. We do this by enforcing the delay as both an invariant on the intermediate, committed node, as well as a guard on the outgoing transition.

allow for modeling I/O by receiving a message and taking a resulting action. Finally, the use of invariants and edge guards enforces that the only exact duration of time expected is allowed.

3.1.3 Firing Automaton

To finish the picture of how I/O works in PyLSE-TA we must describe the issue of output. We will address this issue along with a subtlety unique to SCE. The subtlety is that output pulses take a certain amount of time before being received downstream at another element. During this delay, the emitting element may not be able to receive further inputs without triggering an error. This is difficult to model with a single automaton. Fortunately, TA are closed under a parallel operator, ensuring that two timed automata put into this parallel product behave as one timed automaton. We thus solve our issue by splitting an SCE element into two components, the core machine that tracks state and input, and a separate automaton, the “firing automaton”, that tracks firing and the associated delay. In this way, the original machine can begin responding to new inputs, tracking error conditions appropriately while the firing automaton guides the output. This decoupling allows for the necessary interleavings of input and output pulses and is

drawn in 3.6.

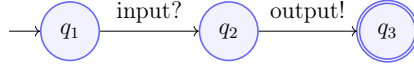


Figure 3.5: I/O is conveyed by actions which decorate transitions. An automaton must use the **receive** action, denoted by a “?” and must use another transition to employ the **send** action, denoted by a “!”. The important distinction from the function picture is that in the case of parallel automata, actions from other automata may interleave between the **receive** and **send** actions.

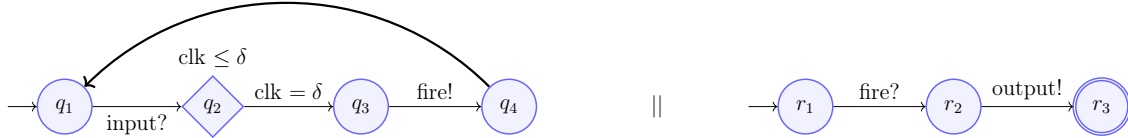


Figure 3.6: We further decouple the output into a firing automaton so that it can cover the propagation delay associated and further allows the core element to continue responding to new inputs without blocking on the propagation delay.

The main difference in Algorithm 3 is that we create a second intermediate node with a transition that sends the output signal. It does so by forwarding the firing signal to a second automaton, the firing automaton, which sits in parallel to the main system. This firing automaton is the component that actually sends the output pulse. In this way, we can allow the main system to continue processing new inputs and changing its own state without blocking on the firing time associated with outputs.

3.1.4 Error Scenarios

With the above structures we have guaranteed that we have described the traces that behave correctly, e.g. in an ideal operation of an SCE element. We must also ensure that incorrect behaviors are made explicit and exposed at a structure level of the

Procedure 3 : expand-trans-firing Further expand an edge that fires.

Input: $ta : \text{PyLSE-TA}$

Output: $\text{edge} \in \text{PyLSE-TA}$

$t_{\text{begin}} \leftarrow \text{create-node}()$

$t_{\text{end}} \leftarrow \text{create-node}()$

▷ create a new, intermediate node which is flagged as committed

$\text{interm} \leftarrow \text{create-node}(\text{committed}=\text{true})$

▷ create a second, intermediate node to send output signal

$\text{fire} \leftarrow \text{create-node}()$

▷ plumb the new states together according to the timing constraints

$\text{create-edge}(t_{\text{begin}}, \text{edge.in-channel?}, \text{clk} \geq \text{edge.delay}_1, \text{clk}, \text{interm})$

$\text{create-edge}(\text{interm}, \emptyset, \text{clk} == \text{edge.delay}_2, \text{clk}, \text{fire})$

$\text{create-edge}(\text{fire}, \text{edge.fire-channel!}, \emptyset, \emptyset, t_{\text{end}})$

▷ create a firing automaton to handle outputs in parallel to the main system

$\text{firing-begin} \leftarrow \text{create-node}()$

$\text{firing-interm} \leftarrow \text{create-node}()$

$\text{firing-end} \leftarrow \text{create-node}()$

$\text{create-edge}(\text{firing-begin}, \text{edge.fire-channel?}, \emptyset, \emptyset, \text{firing-interm})$

$\text{create-edge}(\text{firing-interm}, \text{edge.out-channel?}, \emptyset, \emptyset, \text{firing-end})$

$ta' \leftarrow \text{parallel-produce}([\text{cast-to-ta}(t_{\text{begin}}), \text{cast-to-ta}(\text{firing-begin})])$

return ta'

timed automaton. The errors that we care about are, for the most part, due to timing violations. We track these with guards on the transitions. All signals are guarded by delays, for instance, a transition delay. For instance, if a signal is received, triggering an internal state change, but this occurs before the adequate `setup time` then the system should indicate this by moving to an error state. A similar situation occurs for `hold time`, in which a signal would again indicate state change, but this would conflict with the necessary delay imposed by the `hold time`, thus requiring movement to some error state. The situation is again the same for `transition time`. In each case, we create error nodes for each signal guarded by the associated timing constraint, indicating that if a signal is received *before* the delay, we should move to an error state (and subsequently deadlock the system). This is schematized in 3.7 in the case of transition times. All other errors are accounted for similarly.

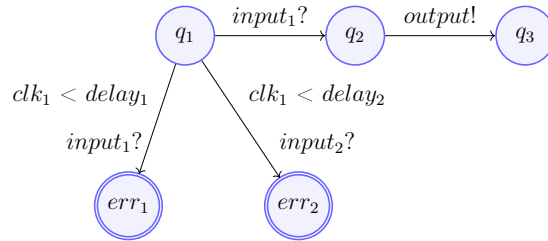


Figure 3.7: In any state that can receive a signal, we create error states which trigger on receipt of the signal when the associated clock is less than necessary transition time.

3.1.5 Putting It All Together

Given a specification of a PyLSE Machine, we translate it into a timed automaton in pieces. We create source and sink automata, iterate over all nodes in the PyLSE Machine specification, expanding each sequence of (source, transition, target) into one that incorporates message passing over channels, creating parallel firing automata as needed. All timing constraints generate error states and appropriate transitions. We

Procedure 4 : create-error-state Create error states for each signal.

Input: circ : PyLSE Machine**Input:** pta : PyLSE Machine**Output:** PyLSE-TA

```

for p ∈ pta.nodes do
  for i ∈ p.inputs do ▷ create an error node and look up the associated transition time delay
    error-i ← create-node()
    error-delay ← circ.p.transition-delay ▷ create a transition from the original node to the
    error state
    create-edge(p, {input-chan}!,  $\text{clk}_i < \text{error-delay}$ ,  $\emptyset$ , error-i)
  return pta

```

offer pseudocode to describe the algorithm in 5.

3.2 Circuit Verification

The PyLSE DSL makes it very simple to rapidly prototype SCE circuits as well as simulate them; however, verification is limited to manually creating and inspecting traces. That is, the designer must a schedule of pulses to input to a design. This input trace generates an output trace which must be inspected to determine if it is correct. Various internal state is similarly dumped in the trace output which can also be inspected for correctness. In this section, we determine what the important properties should be, translate them to TCTL, and then deduce them automatically from a given PyLSE and PyLSE-TA specification. We conclude by summarizing the results presented in the PyLSE paper.

Procedure 5 : PyLSE-to-PyLSE-TA Translate PyLSE Machine to PyLSE-TA.

Input: circ : PyLSE Machine**Output:** PyLSE-TA*▷ Build source and sink automata*

src ← create-source-auto(circ)

snk ← create-sink-auto(circ)

ta ← create-node()

*▷ Expand all transitions in the PyLSE Machine to treat I/O and firing***for** t *in* circ.transitions **do** **if** is-firing(t) **then**

ta ← expand-trans-firing(ta,t)

else

ta ← expand-trans(ta,t)

▷ Create the parallel product of all derived automata

ta ← parallel-product([src, snk, ta])

return ta

3.2.1 Correctness Properties

As noted above, there are two properties we build into the PyLSE-TA automata, aiming to structurally enforce the traces that represent ideal behavior of the circuit and expose the traces that violate timing constraints. We express correctness in terms of a logical formula using Uppaal’s modal logic, TCTL, timed variant of Computation Tree Logic. Formulas in TCTL can reference states in the timed automaton as well as clocks. TCTL, following its parent CTL, is a branching time logic, and so describe traces as trees where each branch is a possible trace originating from a common configuration. TCTL uses two pairs of modal operators, **A** and **E**, and **[]** and **<>**, both of which are connected by a negation operator in a set of De Morgan equalities. **A** indicates “all possible branches” and **[]** indicates “all possible depths (of a branch)”. TCTL additionally has disjunction and conjunction operators, **||** and **&&** respectively, as well as an implication operator, **imply**.

To address the first type of formula, we observe that we can manually create a correct schedule of inputs using the source automaton, as described above. Moreover, we can use the firing automata to ensure output pulses are triggered at the correct times as well. In addition to checking this for each relevant firing automaton, we need to check this for all possible traces. Uppaal uses a variant of Computation Tree Logic (CTL) that is timing sensitive and so we are able to convey all of our requirements directly. For example, if we want to ensure that all possible traces of a system trigger an output at timestamp 4, we would write this as `(A[] fA.end imply global == 4)`, where `fA` is the relevant firing automaton. The symbol `A[]` is in fact two modal operators, **A** which means “for all possible branches” and **[]** which means “for all possible depths of a branch”. `fA.end` is a state which triggers an output and `global` is the overarching clock in the system which Uppaal uses to keep track of absolute time. In totality, the formula expresses correctness

by asserting that in a system's final configuration, we triggered an output at the correct time under what we have deemed a correct input sequence. This is written more formally in Definition 7.

Definition 7 (Successful Termination)

A PyLSE-TA is said to terminate successfully if the following property holds for the initial state.

$$A[] \bigwedge_{i \in S.firing-auto} (S.fA_i \supset (\bigvee_{t \in valid-end-times} global-clock = t))$$

The second type of correctness formula is even simpler, as it asserts that we simply never reach an error state. The formula can be written as $(A[] \text{ not } (S.err1 \ || \ S.err2 \ || \ \dots))$ where S is an automaton with error states. This formula uses the same modal quantifier as in the first, scoping all possible traces of all possible depths, and then simply enforces that in all of these situations, we never trigger any of the error states. A way to think about this formula is that it guarantees the system does not deadlock, e.g. fall into an error state. This is defined more formally in Definition 8. As we have built in all possible error states, this formula gives us confidence that the circuit element is correct, both in that it does what we expect, and doesn't do what we don't expect.

Definition 8 (No Deadlock)

A PyLSE-TA is said to enjoy deadlock freedom if the following property holds for the initial state.

$$A[] \neg(\bigvee_{i \in S.error} S.err_i)$$

The above properties can moreover be generated automatically based on the information in the PyLSE Machine specification and the automata data generated in the PyLSE-TA translation. The first type of formula depends on the firing automata and their clocks while the second depends on the error states. These are both accessible in

the final automaton and thus may be used to incrementally build the corresponding correctness formulas.

3.3 Results

We reproduce the table shown in [1] to demonstrate the results of the PyLSE-TA translation and verification scheme. Table 3.1 shows that it is possible to automatically verify correctness properties of several core SCE elements. The results specifically demonstrate that we can be confident in the designs of the core library of elements, for instance the destructive read-out (DRO), C-element, etc. We moreover, are able to verify slightly larger circuits, including a comparator and bitonic sorter. Nevertheless, the table also demonstrates the limits of the effectiveness of this approach, as circuits any larger than the above time out. This underlines a common pitfall in the field of model checking, namely state space explosion. The number of states in a timed automaton scales exponentially with the number of clocks, and unfortunately, SCE makes extensive use of clocks. The conclusion is that to scale verification, we must look elsewhere for more adaptable approaches. The lessons we take with us are that the above approach is tightly wedded to a concrete model, and that composition of models is large and complex.

3.3.1 Discussion

We have described PyLSE-TA, a verification language that uses timed automata as its underlying formalism. By translating PyLSE machines into TA we obtain a formal model capable of defining circuit correctness in the form of TCTL formulas as well as a way to mechanically check whether designs conform to the specifications. PyLSE-TA puts PyLSE machines on a firm foundation and gives us greater confidence in our ability to rapidly prototype designs. Nevertheless, timed automata suffer from state space

Name	PyLSE				UPPAAL						Comparison		
	Size	Cells	States	Tran.	TA	Locs.	Tran.	Chan.	Time (s)	States	TA/Cells	Locs./States	Tran.(U)/Tran.(P)
C	6	1	3	6	2	39	42	3	<1	38	2	13	7
InvC	6	1	3	6	4	45	48	3	<1	69	4	15	8
M	2	1	1	2	2	17	18	3	<1	37	2	17	9
S	1	1	1	1	3	13	13	3	<1	56	3	13	13
JTL	1	1	1	1	2	9	9	2	<1	17	2	9	9
And	11	1	4	12	5	102	110	4	<1	69	5	25.5	9.17
Or	4	1	2	6	2	49	53	4	<1	48	5	24.5	8.83
Nand	12	1	4	12	2	95	103	4	<1	42	2	23.75	8.58
Nor	6	1	2	6	2	49	53	4	<1	36	2	24.5	8.83
Xor	9	1	3	9	3	75	81	4	<1	45	3	25	9
Xnor	12	1	4	12	2	94	102	4	<1	45	2	23.5	8.5
Inv	4	1	2	4	3	30	32	3	<1	14	3	15	8
DRO	4	1	2	4	2	27	29	3	<1	11	2	13.5	7.25
DRO SR	6	1	2	6	2	49	53	4	<1	23	2	24.5	8.83
DRO C	4	1	2	4	3	31	33	4	<1	14	3	15.5	8.25
2x2 Join	20	1	5	20	5	206	221	8	<1	58	5	41.2	11.05
Min-Max	5	5	9	15	24	149	155	14	<1	2471	4.8	16.56	10.33
Race Tree	16	18	32	56	50	440	464	54	127/84	262559	2.78	13.75	8.29
Adder (Sync)	13	19	33	71	57	627	665	62	669/515	1858153	3	19	9.37
Adder (xSFQ)	31	83	121	183	193	1449	1511	211	∞	N/A	2.33	11.98	8.26
Bitonic Sort 4	6	30	54	90	144	894	930	84	∞	N/A	4.8	16.56	10.33
Bitonic Sort 8	24	120	216	360	576	3576	3720	336	∞	N/A	4.8	16.56	10.33

Table 3.1: The UPPAAL column shows that model checking proves sufficient (and efficient) for verifying the positive and negative correctness properties of the basic SCE cells. It also manages to work well for the small designs of the Race Tree and Min-Max. We can see however that it becomes infeasible on slightly larger designs, namely, the Adders and Sorters.

explosion, imposing a formidable limit on model checking. This leaves us with the ability to only verify relatively small designs. We are inspired to look into alternative ways to model PyLSE machines. In particular, functional aspect of PyLSE Machines and their ease of composition is somewhat lost at the level of automata. We use this as a starting place for further investigation.

Chapter 4

Programming Machines, Coinductively

In this chapter we introduce the second language, Citrus, a DSL shallowly embedded in the Agda theorem prover language. Citrus constructs a formal semantics for PyLSE Machines by mapping them onto stateful, reactive functions in a coinductive type system. This allows Citrus to recover the compositionality inherent in SCE circuits, greatly simplifying the construction of circuits, compared to PyLSE-TA, but maintaining a strong formalization that enables verification. Specifically, composing timed automata is a costly, monolithic procedure. Properties defined for one automaton do not naturally adapt or translate when it changes, and composing corresponding properties does suffers a similar problem. On the other hand, observing that the SCE circuits have simpler compositionality constraints allows us to give a much more algebraic description of the circuits. In terms of verification, using a coinductive type system turns verifying correctness properties into a matter of typechecking, as opposed to state space exploration as in model checking. Additionally, Citrus expands on the verification capabilities by formalizing a notion of equivalence for SCE circuits.

The semantics of Citrus is inspired by functional reactive programming (FRP). FRP is a style of programming in which responsive, stateful elements are composed in dataflow

graph-like networks. Arrows are a specific such element that can be thought of as generalizations of functions. Citrus translates SCE circuits into a type of arrows that captures the looping and statefulness of the underlying circuits, as well as provides the combinators necessary to arrange them into larger circuits. The utility of coinductive types specifically can be seen from a logical perspective. We would like to write properties that involve infinite behavior of the circuits as opposed to just arbitrarily finite traces. Coinduction, as a dual to induction, offers proof principles for reasoning about non-termination in this sense. Simply put, coinduction is a natural match for the “always-on”, looping quality of compositional, reactive elements.

4.1 The Citrus Language

Citrus is an embedded language inside Agda, consisting of abstractions and operators aimed at describing SCE circuits as PyLSE Machines. These abstractions and operators are: (1) core data structures for representing timed pulses; (2) the `PulseGate` data structure which represents basic SCE elements; and (3) the composition operators used to compose `PulseGates` into SCE circuits.

4.1.1 Timed Pulses

To model SCE circuits, we must model infinitesimal pulses occurring at specific timestamps. At a given timestamp we can observe a “pulse”, which is either an instantaneous packet of information or its absence. For modeling purposes we add a third possibility, an “error” signal used to model when a pulse arrives at a gate during a time interval in which the gate cannot receive a pulse correctly (and which is propagated as the output of the gate when this event happens). While SCE circuits notionally operate in real-valued time, Citrus uses the natural numbers instead—all time intervals that we deal with are

rational, and by choosing the proper units we can describe them using naturals. Pulses in SCE circuits travel on wires, and we abstract these wires as vectors of pulses (with one element per wire). We can now describe our pulse-based I/O representation as shown in Listing 4.1.

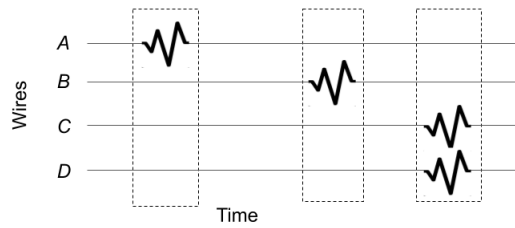


Figure 4.1: We treat pulses in Citrus as “vertical” slices of time across a selection of wires. Each slice is a vector which is indexed by the number of wires. Here we have three `TimedPulse` 4s.

```
Time : Set
Time = ℕ
data Pulse : Set where
  * : Pulse -- pulse
  ∅ : Pulse -- no pulse
  err : Pulse -- error signal
TimedPulse : (n : Time) → Set
TimedPulse n = Vec Pulse n × Time
```

Listing 4.1: The basic definitions for dealing with time in Citrus: time is a Natural number; a `Pulse` is one of three possible outcomes; a `TimedPulse` is a vector of `Pulses` indexed by the number of wires on which a pulse can arrive.

As the listing shows, a Citrus program will both take and output a vector of pulses at a given timestamp. A `Pulse` itself is one of three possible outcomes: a pulse, nothing, or an error signal. A `TimedPulse` is a vector of `Pulses` indexed by the number of wires on which a pulse can arrive. This “vertical” view of time means that we do not explicitly keep track of a history of pulses. This representation tracks the operation of SCE more

naturally than the alternative because SCE gates themselves keep track of I/O and state information in an asynchronous, non-global manner.

4.1.2 PyLSE Machines as Coalgebras

The PyLSE Machine is itself based on a type of finite state transducer called a Mealy Machine. Mealy Machines can be thought of as stateful functions, or maps between finite state automata. More formally, Mealy Machines have a straightforward presentation in terms of coalgebras [42]. The theory of coalgebras has been suggested as a natural framework for state-based systems: while terminating functions are usually represented as recursive/inductive objects, state-based, non-terminating systems are represented using corecursion and coinduction instead. The former can be presented as algebras and the latter as coalgebras. We do not require the details of this theory for our purposes and direct the interested reader to [43]. Fortunately, the theorem prover Agda makes available a wide class of coinductive types using the `record` structure. The basic Mealy Machine coalgebra is interpreted by the coinductive type defined in 4.2 below.

```
record MealyMachine (I O : Set) : Set where
  coinductive
  field
    resp : I → O
    step : I → MealyMachine I O
```

Listing 4.2: MealyMachine data structure

The interpretation of Listing 4.2 as a record is that, given a term of this type, we can *deconstruct* it using its fields. That is, given a term of type `MealyMachine`, we can call the `resp` and `step` functions on it. The `resp` function stands for “response” and, given an input of type `I`, will give us an output of type `O`. The `step` function takes an input of type `I` and emits the next step of the object “in time”. The keyword `coinductive` tells the type checker that we can do this infinitely. The text can be intuitively restructured

as follows: $MealyMachine \rightarrow I \rightarrow O \times MealyMachine$, i.e., given a `MealyMachine` and an input, we can generate an output and the next `MealyMachine`. For `Citrus`, the inputs I and O should be `TimedPulses`.

The `MealyMachine` type will serve as a basic building block for our final `PulseGate` type, which we will use to model SCE circuits; however, currently it is only able to reflect the I/O behavior of a gate, e.g., the `TimedPulse` information. In reality, each SCE gate additionally keeps track of unique, internal state. For example, the LA and FA gates each keep track of their own distinct information on different transition delays and timing constraints. We model this information with the type `StateVector` in Listing 4.3. The `StateVector` keeps track of all internal state for a given SCE element. For the most part, we use the definitions provided in the PyLSE paper, making small simplifications. The PyLSE Machine needs to keep track of roughly the following state: (1) current state; (2) each input’s last arrival timestamp; (3) time required to transition from one state to another given an input; (4) time taken for an output to be observable; (5) timestamp of the end of the unstable period; (6) various timing constraints related to hold/setup time.

Now that we have the type of internal state represented, we must decide how best to incorporate it into the basic `MealyMachine` type. We *could* expose a gate’s internal state by lifting the state into its type signature as we showed for `MealyMachine`, i.e., $I \rightarrow S \rightarrow O \times S$ where S is the type of the state, but then, as a practical matter, it becomes difficult to write generic composition functions. That is, gates should only be composable in terms of their I/O behavior (i.e., the pulses sent and received)—each gate’s internal state should be hidden from the other gates they are composed with. We need an individual gate’s state to be encapsulated so that it does not appear in that gate’s type-level signature. When the overall circuit (the composition of gates) is stepped forward in time, each gate’s state should be automatically (and invisibly) looped back to itself to advance and maintain their individual states as appropriate. The `MealyMachine`

```

record StateVector (X : Set) (n : ℕ) : Set where
  constructor ⟨_,_,_,_,_,_,_⟩
  field
    -- current state
    qcurr      : X
    -- when did i last see this input?
    Ω          : Vec Pulse n → ℕ
    -- from a given state and input, how
    -- long does it take to update?
    tmap       : X → Vec Pulse n → ℕ
    -- tarr + ttrans
    tdone      : Time
    -- same as tmap but for outputting
    firing-delay : Time
    -- setup time constraints (list of
    -- deadlines for given pulses)
    θ : (X × (Vec Pulse n)) → List ((Vec Pulse 2) × Time)

```

Listing 4.3: The StateVector data structure used to hold timing and node information. The list θ contains the *setup time* constraints which are timestamps beyond which receiving inputs would result in a violation.

```

initDataC : StateVector S3 2
qcurr initDataC = s30
Ω initDataC = λ x → 0
tmap initDataC = λ x v → 0
tdone initDataC = 0
firing-delay initDataC = 8
θ initDataC = λ (x,y) → []

```

Listing 4.4: The data that defines and initializes the LA gate. The type S_3 is used to represent unique internal state of the LA gate.

type does not fit this requirement (because it exposes the state information), and so we need to extend our definitions further to get a suitable `PulseGate` definition.

To implement the desired behavior we turn to a concept explored in functional reactive programming, namely that of a *loop*, or *trace* [44]. Several structures in functional programming permit the construction of a *loop* operator. The basic type signature takes a function of type $I \times S \rightarrow O \times S$ and turns it into one of $I \rightarrow O$, that is, it invisibly feeds back the S parameter yielding a function that only exposes its I/O parameters. The *loop* operator for `PulseGates` can be implemented as below in Listing 4.5, where the type S is the type of internal state being “fed back”.

```

loop : (fdbk : S) →
      (m : MealyMachine (TimedPulse n1 × S) (TimedPulse n2 ×
        S)) →
      PulseGate (TimedPulse n1) (TimedPulse n2)
resp (loop fdbk m) tp = proj1 (resp m (tp , fdbk))
step (loop fdbk m) tp = loop (proj2 (resp m (tp , fdbk)))
                           (step m (tp , fdbk))

```

Listing 4.5: The Trace operator implemented for the `PulseGate` data structure.

We briefly call attention to the two equations that define the `loop` function. Contrary to typical Haskell-style definitions, the record fields of the `PulseGate` are called on the function itself. This is a corecursive function as opposed to a recursive one. The interpretation is that given some term `loop fdbk m` and an input `tp` (a `TimedPulse`), calling the `resp` function on this structure should yield the right-hand side of the equation. The function is defined corecursively, that is to say, it is defined by what the destructors of the record do to terms of the given type. With these definitions in place we can describe the `PulseGate` data structure.

Definition 9 (`PulseGate`)

The `PulseGate` is a looped `MealyMachine` with I/O given as `TimedPulses` and feedback information corresponding to `StateVectors` representing a specific `SCE` gate’s unique

internal state.

To put Definition 9 into practice, we construct an example corresponding to the Last Arrival gate from Figure 2.1. We proceed in two steps. In the first step we create a function, the underlying transducer, that exposes the internal state information, and use it to form a `MealyMachine`. In the second step we apply the `loop` operator to this machine and its associated initial data to obtain the full circuit. The underlying transducer of the Last Arrival gate is pictured below, in abbreviated form in Listing 4.6. It defines how the gate behaves according to pulse inputs and how internal state, such as its own clock and timing constraints, are updated.

```

LA-transducer : TimedPulse 2 × StateVector S3 2 → TimedPulse
  1 × StateVector S3 2
LA-transducer ((v,tarr), ⟨qcurr, Ω, tmap, tdone, fd, θ⟩) with
  (validInput tarr tdone)
LA-transducer (v@(*::*::[], tarr), ⟨s30, Ω, tmap, tdone, fd, θ
  ⟩) | true =
  ((*::[]), (tarr + fd)), ⟨ s30 , (λ s → tarr), tmap, (tarr +
  fd), fd, θ⟩
...
LA-transducer (v@(*::∅::[], tarr), ⟨ s30, Ω , tmap, tdone , fd
  , θ ⟩) | true =
  ((∅::[]), (tarr + tmap s30 (proj1 v))), ⟨s30 , (λ s → tarr),
  tmap, (tarr + tmap s30 (proj1 v)), fd, θ⟩
...
LA-transducer ((v, tarr), ⟨ qcurr, Ω, tmap, tdone, fd, θ ⟩) |
  _ =
  ((err::[]), (tarr + tmap qcurr v)), ⟨s3e, (λ s → tarr), tmap,
  (tarr + tmap qcurr v), fd, θ⟩

```

Listing 4.6: A few example lines from the definition of the transducer function for the LA gate

The function pattern matches on the current state *qcurr* (in the sense of a node in the PyLSE Machine graph) and the input slices, which can consist of a pulse on each wire, a pulse on no wires, or pulses on one or the other wires. There can also be error signals ingested which simply propagate through the circuit as a whole. We follow PyLSE in that we assume pulses are tagged with an absolute timestamp, as opposed to a relative

timestamp. The `StateVector` then keeps track of the timestamps of each I/O pulse, as well as when the element is ready to accept new inputs. The syntax “`with (validInput tarr tdone)`” conveys that we should only begin computing the transition if the given input does not conflict with the element in any way. The function *validInput* is based on the PyLSE paper’s *ERROR-K-TRAN* and *ERROR-K-CONS* rules; intuitively it sends to an error state (and emits an error signal) if the timing constraints are violated in some way. In all then, the way to interpret the `LA-transducer` function is to look at what kind of `TimedPulse` arrived, what internal state the transducer was holding onto, compute the next state, whether or not there is an output pulse, and then the times taken for transitioning or firing and when the next stable period begins.

We can now proceed to the second step which combines the underlying transducer with its initial state information under the `loop` operator and is pictured in Listing 4.7. This data structure satisfies all of our main requirements: At the type level, it exposes only the I/O behavior of the circuit, allowing it to be more easily composed; it keeps track of unique internal state via the underlying transducer function; and it runs infinitely, always able to respond to new inputs (modulo error-inducing inputs).

```
-- the Last Arrival gate
LA : PulseGate (TimedPulse 2) (TimedPulse 1)
LA = loopD initDataC LA-transducer
```

Listing 4.7: The full Last Arrival gate as represented in the Citrus language. The `loopD` is a small variant of the `loop` function presented above that casts the transducer into a `MealyMachine`.

4.1.3 Composition Operators

Functional arrows benefit from several composition operators that allow one to construct large dataflow-like graphs. We extend our `PulseGate` structures with a handful of such composition operators. We have defined operators to put `PulseGates` in parallel, in

sequence, and to combine a pair of PulseGates but propagate only the behavior of the first; together these operators are sufficient to define a wide array of possible compositional behaviors and give the user ample capability to design SCE circuits as if they were dataflow-like networks. This design improves upon PyLSE itself in that the composition information is represented at the type level, which facilitates easier debugging as well as easier programming. Below we describe the sequence and parallel composition operators as examples. The sequence operator (Listing 4.8) funnels the output of one PulseGate as the input to the next. The parallel composition operator (Listing 4.9) is capable of processing inputs simultaneously or, if inputs come only on a single wire, one at a time as they are ingressed.

```
_>>>_ : (m1 : PulseGate I M) → (m2 : PulseGate M O) → PulseGate I O
resp (m1 >>> m2) = resp m2 ∘ resp m1
step (m1 >>> m2) = λ x → (step m1 x) >>> (step m2 (resp m1 x))
```

Listing 4.8: The sequence operator chains two PulseGates one after the other.

```
_***_ : (m1 : PulseGate (TimedPulse i) (TimedPulse j)) →
        (m2 : PulseGate (TimedPulse m) (TimedPulse p)) →
        PulseGate (TimedPulse (i + m)) (TimedPulse (j + p))
resp (_***_ m1 m2) = λ vim → resp m1 (proj1 (splitAtTP i
    vim))
    ++TP resp m2 (proj2 (splitAtTP i vim))
step (_***_ m1 m2) = λ vim → step m1 (proj1 (splitAtTP i
    vim)) ***
    step m2 (proj2 (splitAtTP i vim))
```

Listing 4.9: The parallel composition operator. The indices i, j, m, p are natural numbers indicating widths.

The composition operators do not impose any timing penalties beyond those imposed by the gates being composed, that is, they switch instantaneously. Finally, we note that networking circuits together frequently requires operations like duplicating and forking the wires. As an example, we present the “cloning” function below which duplicates one of its input wires in a specific order. In Citrus these “wire management” functions are

turned into `PulseGates` and then composed using the same composition operators.

```
cloning : TimedPulse 3 → TimedPulse 4
cloning (a :: b :: c :: [] , t) = (a :: b :: a :: c :: [] ,
  t)
```

Listing 4.10: A simple function which duplicates its input in a specific pattern. Functions like these are necessary to do so-called “wire-management” in larger designs.

4.1.4 Programming with PulseGates

In this section we present a basic example of programming with the Citrus framework. We use as our running example the `min-max` element, or comparator. It is visualized in Figure 4.2 and displayed as Citrus code in Listing 4.11. The comparator takes two inputs and pulses its first wire on the earliest input pulse, and its second wire on the latest input pulse. This has the effect of sorting the inputs from earliest to latest. We can see the close correspondence between the visual description of the SCE circuit and the Citrus version as code.

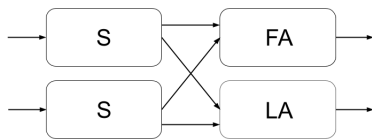


Figure 4.2: The comparator visualized as a network of SCE elements.

```
min-max : PulseGate
  (TimedPulse 2)

  (TimedPulse 2)
min-max = (S *** S) >>>
          ((arr perm0) >>
           >
           (FA *** LA))
```

Listing 4.11: The comparator element as a term in Citrus. `perm0` is a wire management function and `arr` promotes it to a `PulseGate`.

Finally, we give some examples of running simulations of the above circuits. We first define a few sample inputs shown in Listing 4.12. For instance, the first input is on the

second wire. We submit these inputs to the comparator circuit and observe the outputs, as shown in Listing 4.13.

```
-- a pulse on the second wire at time 4
test-pulse-1 : TimedPulse 2
test-pulse-1 = ( $\emptyset$  :: * :: []) , 4

-- a pulse on the first wire one time point later
test-pulse-2 : TimedPulse 2
test-pulse-2 = (* ::  $\emptyset$  :: []) , 5

-- a pulse on the first wire 9 (8) time units later
test-pulse-3 : TimedPulse 2
test-pulse-3 = (* ::  $\emptyset$  :: []) , 13
```

Listing 4.12: Test Inputs

```
-- * ::  $\emptyset$  :: [] , 13
out= proj1 ((runPG min-max) test-pulse-1)
nxt= proj2 ((runPG min-max) test-pulse-1)
-- err :: * :: [] , 10
out'= proj1 ((runPG nxt) test-pulse-2)
--  $\emptyset$  :: * :: [] , 18
out''= proj1 ((runPG nxt) test-pulse-3)
```

Listing 4.13: Comparator Test

Figure 4.3: Test inputs are simple TimedPulse objects or vectors of pulses/absences/errors at specific timestamps. Running a circuit against a sequence of test inputs involves the helper function `runPG`. We show in comments the output of running the functions.

We need to submit both inputs to the comparator to fully realize its behavior. We can see from `out` that the behavior of the comparator correctly pulses its first wire at the input timestamp plus its internal transition time/firing delay. During this period, receiving another input would be an error, as shown by result `out'`. However, the third input `test-pulse-3` is sufficiently far in time and can trigger the comparator completely. That is, the comparator pulses its second wire appropriately delayed in time. This example demonstrates that Citrus allows SCE developers to simulate the semantics of SCE circuits for testing and prototyping in a simple, functional style.

4.2 Equational Reasoning

Now that we have a formal representation of `PulseGates` and of SCE circuits as compositions of `PulseGates` (i.e., as `PyLSE Machines`), we turn our attention to the notion of equivalence between SCE circuits. That is, when are two `PyLSE Machines` behaviorally equivalent, or in other words, when do two machines yield the same observable outputs given the same inputs? To this end, we adapt the notion of bisimulation and discuss how it may be used to prove the equivalence of two `PyLSE Machines` in Section 4.2.1. We discuss two finer points related to these notions in Section 4.2.2 which are required to fully complete our proof technique.

4.2.1 Equivalence between `PulseGates`

The classical treatment of equivalence for reactive systems is in terms of bisimulation over infinite labeled transitions systems (LTS) [19]. Because the LTS do not have a termination condition the equivalence between them cannot depend on recursing to a base case (which computes a least fixpoint); instead it must depend on the greatest fixpoint [45]. To establish bisimilarity in this framework we check: (1) that if one LTS can make a transition α then the other LTS can make a transition α ; and (2) the resulting states of the two LTS are bisimilar. Mathematically, this is a coinductive relation, and its coinductive nature is captured clearly by the `PulseGate` coinductive records that we use to describe SCE circuits.

The framework of bisimulation has been extended to cover time-sensitive LTS, such as timed automata. There are several different notions of timed bisimulation [19], including “timed bisimulation” and “time-abstract bisimulation”. The former requires that the two LTS must make the same transitions with the same delays, which is too restrictive for our purposes (e.g., proving that one SCE circuit is an optimized version of a second, with

equivalent behavior but smaller delay). Time-abstract bisimulation, however, turns out to be a good choice to prove circuit equivalence. In this setting the two LTS must take the same transitions but are allowed to do so with differing delays. Intuitively, it says that for two SCE circuits to be time-abstract bisimilar there must be some amount of time they each delay (which can be different for each one) such that, given the same input sequence, they must respond with the same output sequence. Note that time-abstract bisimilarity subsumes weak bisimilarity (which allows an LTS to take “silent” transitions) because a silent transition is essentially equivalent to an additional delay.

This notion of bisimilarity works well for SCE circuits because SCE circuit semantics depend on the timing of pulses *relative* to the circuit characteristics such as transition and firing delays, rather than absolute timing. Time-abstract bisimilar circuits thus have the same “cycling” behavior—that is, if we use the Mealy Machines in Figure 2.1 as a convenient reference, an input sequence that causes one circuit to go from its *Init* state through either the *a arrived* or *b arrived* states and then back to *Init* (respecting its various time delays) will also cause another bisimilar circuit to do the same (respecting *its* various time delays).

We begin with bisimilarity as it is presented in Abel’s work on formal languages [46] which we reproduce in Listing 4.14, specialized to `PulseGates`. This type is a coinductive record called $_ \approx \langle _ \rangle \approx _$, where the first and last underscores stand for the circuits being compared and the middle underscore stands for the depth of observation, e.g., $A \approx \langle \infty \rangle \approx B$ means that A and B are time-abstract bisimilar for arbitrary depths of observation. We direct the reader to the `eqr.agda` file in the supplementary material for the full development; here we show a simplified presentation in Listing 4.14 removing the size annotations for readability. The record takes two `PulseGates` as inputs and has two fields, `≈resp` and `≈step`. The first, `≈resp`, represents a proof that the outputs of the two gates are equivalent; the second, `≈step`, establishes that the stepped gates

(yielded as a result of a `PulseGate step`) also bear the equivalence relationship. In more traditional terminology, these two fields state that the heads of the two gates are equal and the tails are (time-abstract) bisimilar.

```

record  $\approx\langle \rangle \approx$  {A B : Set} (m1 : PulseGate A B) (m2 :
  PulseGate A B) : Set where
  coinductive
  field
     $\approx \nu$  : resp m1  $\equiv$  resp m2
     $\approx \delta$  :  $\forall$  (a : A)  $\rightarrow$  (step m1 a)  $\approx\langle \rangle \approx$  (step m2 a)
    
```

Listing 4.14: The bisimilarity relation follows the definition pattern we used for `PulseGates`. A proof that two `PulseGates` are equivalent requires a proof that all inputs are equal (this is the $\equiv$ symbol) for arbitrary delays and that for all inputs after arbitrarily delaying the continuations of the gates are also equivalent.

The relation in Listing 4.14 is an equivalence relation, satisfying the properties of reflexivity, symmetry, and transitivity. Proofs of these facts are straightforward and based on simple copattern matching. However, this current definition is incomplete because it does not incorporate any notion of *time*. In order to address this issue we must explain how we can reason about sequences of pulses over time.

4.2.2 Ordered Sequences and the Coinduction Hypothesis

There are two important issues that remain to be solved before we can prove two SCE circuits as equivalent. The first problem is that we need to establish that for all possible sequences of inputs (recalling that an input is a vector of pulses, one per wire, plus a timestamp), two circuits output the same thing; however, “all possible inputs” is too many to feasibly check.

The stateful semantics of SCE gates provides us with an answer: an SCE gate is assumed to start in some initial state, then to get at least one pulse on each wire (cycling through some number of states internal to that gate) before returning back to the initial state. For example, the Last Arrival gate and First Arrival gate from Figure 2.1 both

have two input wires (represented as possible inputs **a** or **b**) and, once having received a pulse on some wire, cannot return to their initial states before receiving an input on the remaining wire; the different is that Last Arrival only fires when it receives a pulse on the remaining wire that hasn't seen a pulse yet, while First Arrival fires immediately upon receiving the first pulse. Let us call the period from when a gate is in its initial state and receives some pulse to the point it has received a pulse on all input wires and returned to its initial state a *regime*. Then we only need to check sequences of inputs that advance through a regime, completing a gate's cycle of behavior. In particular, we can ignore the possibility of any inputs that advance a gate's PyLSE Machine along a reflexive edge with no output, because that input has no effect on the gate's behavior—for example, if a Last Arrival gate is in state “**a arrived**” and receives another pulse on the **a** gate, that input has no effect and we don't need to consider it when establishing equivalence. We summarize the situation as follows.

- **Repeated pulses.** It's possible for multiple pulses to occur on a single wire without any intervening pulses on the other wires. For all the gates we consider, the setup time constraints, θ , are empty and so we can be confident that a repeated pulse leaves Circuit1 and Circuit2 in the same configuration as before the repeated pulse, and hence we can ignore this possibility when enumerating cases. There is an issue with timing, as repeated pulses will in general update the internal clocks of SCE gates. We treat this issue in more detail below.
- **Simultaneous pulses.** Simultaneous pulses on two wires could occur in theory (though no known SCE gates allow this in reality). The behavior of simultaneous pulses on wires x and y would be either the behavior of receiving x then y or the reverse; which one is unimportant since we cover both of these cases in the proof.

The timing component of this case poses no problem as the circuit essentially bypasses the “dirty” state and returns to its “clean” initial state immediately. That is, there is no risk of getting stuck in an internal state and then violating a timing constraint.

As mentioned in the case of repeated pulses, we must expand on the timing implications of this approach to proving equivalence. We are interested in two SCE gates simulating each other’s I/O behavior up to certain time delays. We are satisfied, then, if one circuit delays for awhile and then mimics the other circuit’s behavior; in fact, this could be a sign that one circuit is some optimized version of the other. Nevertheless, it could be that a circuit delays “too much” and either one machine risks receiving a pulse that violates an internal timing constraint, or the two circuits somehow fall out of (delayed) lock step. The approach we take to remedying this problem is to consider only a representative sequence of input pulses. We know in advance when a sequence of timed pulses will trigger a timing violation. In fact, we can compute this information *a priori* from the initial data or we can interactively determine it by running a circuit against inputs and checking the outputs. The first approach lends itself to automation and while we do not provide that facility in this paper, we do leave it to future work. Given a sequence of inputs that are correctly timed, we know the gate will conform to its intended functionality. It is sufficient then to test circuits for equivalence up to these “well-timed” sequences. To make this clearer, we demonstrate in Listing 4.15 and Listing 4.16 the two concepts that underly the above discussion of time. The first shows how we represent and feed input sequences to `PulseGates` while the second describes the “cyclic” property of SCE gates according to a coinduction hypothesis.

By collecting the distinct input sequences together we are able to run the machine through the sequences to determine their complete profile of behavior. It is the outputs

```

let (FA-out-1 , FA-next-1) = runPG FA    (inp-A t1)
    (FA-out-2 , FA-next-2) = runPG FAcomm-next-1 (inp-B t2)
    (FAcomm-out-1 , FAcomm-next-1) = runPG FA    (flip (inp-A t1))
    (FAcomm-out-2 , FAcomm-next-2) = runPG FAcomm-next-1 (flip
        (inp-B t2))
in (FA-out-1 ~ FAcomm-out-1) ×
   (FA-out-2 ~ FAcomm-out-2) ×
   (FA-next-2 ~⟨ ⟩~ FAcomm-next-2)

```

Listing 4.15: We specify inputs by *sequences* of pulses for instance `inp-A` and `inp-B`. We then run the circuits on these inputs and keep track of the inputs and outputs.

```

coind : proj2 (runPG
  (proj2 (runPG FA (inp-A t1))) (inp-B t2))
~⟨ ⟩~
proj2 (runPG
  (proj2 (runPG FA
    (∅ :: * :: [] , proj2 (inp-A t1))))
    (* :: ∅ :: [] , proj2 (inp-B t2)))

```

Listing 4.16: A coinduction hypothesis is required to validate the basic behavior of all the SCE gates. This hypothesis corresponds to the cyclic behavior that is unique to all SCE gates.

along the way that need to be equivalent and the stepped machines that must be bisimilar. This proof setup brings up the last remaining problem to actually proving two circuits bisimilar: The PulseGates are coinductive and able to be “stepped” forward infinitely, indeed modeling the real life SCE gates; but when can we safely stop for the purposes of our proof? We take advantage of the built-in notion of “cyclicity” that guarantees well-designed SCE circuits will start in some initial, ready state, receive inputs, perform some work, and then eventually return to the initial state. This cyclicity is in reality what we are capturing with our coinductive structures, not solely the ability to be stepped forward infinitely.

To put this in perspective, we can see in Listing 4.15 that after supplying both inputs to the FA gate, we could simply keep doing this, stepping the circuit repeatedly with new inputs. This would be strange however, because we know that the function of the FA gate is to return to its initial state, having performed its output, at the completion of this

sequence. We would like to somehow wrap up this observation and use it to complete the proof. In the setting of coinductive proofs, this type of maneuver is called a “coinductive hypothesis”, dual to an inductive hypothesis. The coinductive hypothesis captures a cycle of behavior that appears in the otherwise infinite stepping of a coinductive object. Thus, in the Citrus language, every basic SCE gate comes with a coinductive hypothesis (derived directly from the intended behavior of that gate) establishing the correctness of the behavior of the circuit. Put differently, we are able to build SCE gates in a “correct by construction” fashion. An example of the coinductive hypothesis corresponding to the FA gate above and the given input sequence is shown in Listing 4.16. With all of this in place we are able to more formally state the definition of `PulseGate` equivalence. These are guidelines for the proof technique, governing the examples, and note present as types in the code. We begin with the concept of the ordered sequence that does not trigger timing violations.

Definition 10 (Well-timed Sequence)

*For all $i, j : \mathbb{N}$, let s be a sequence of type $(\text{TimedPulse } i)$ and m a `PulseGate` $(\text{PulseGate } i) (\text{TimedPulse } j)$, then s is well-timed if for all timestamps in the sequence s , (*resp* m s) does not trigger an error signal.*

This definition is saying that while applying an input to a `PulseGate`, the outputs generated (by feeding the output into the `step` function to continue the circuit) do not trigger an error signal. Currently, we determine well-timed sequences manually, however in future work we intend to automate their discovery. With the definition of well-timed sequences and bisimilarity as a coinductive relation, we present the definition of `PulseGateE` equivalence.

Definition 11 (`PulseGate` Equivalence)

*For all $i, j : \mathbb{N}$, $m_1, m_2 : \text{PulseGate } (\text{TimedPulse } i) (\text{TimedPulse } j)$, (*inp*: $\text{TimedPulse } i$), m_1 is equivalent to m_2 if there is a well-timed sequence s and time delays d_1, d_2 , such*

that for all intermediate outputs (until the coinduction hypothesis is reached):

1. $(resp\ m_1\ s) \equiv (resp\ m_2\ s)$
2. $step_k(m_1) \approx \langle \rangle \approx step_k(m_2)$

where $step(k)$ represents the number of steps that must be applied to the circuit m before it can invoke its coinduction hypothesis and return to its clean state.

Having defined `PulseGate` equivalence we are able to set on a firm foundation many equational laws which SCE circuits obey but which have yet to be proven formally. We pursue two sets of case studies along these lines in Section 4.3.

4.3 Case Studies

We demonstrate the usefulness of Citrus by using it in a set of case studies. The first set of case studies (Section 4.3.1–Section 4.3.3) examine several “laws” about SCE gate composition that depend on equivalences that SCE designers believe should be correct, but that have never formally been proven. The last case study (Section 4.3.4) examines a non-trivial SCE circuit taken from the PyLSE paper [1], a bitonic sorter, and proves a property about it which PyLSE is unable to prove via model checking (i.e., by translating the PyLSE Machine to a timed automata and using a third-party model checker). All of these proofs have been mechanized and are available in the supplementary material.

4.3.1 Temporal Invariance

The first SCE composition law states that delaying two inputs by k time units and running them into a First Arrival gate is the same as running two inputs into a First Arrival and then delaying by k time units. Stated algebraically, where D is a delay and FA is a First Arrival gate:

Proposition 4.3.1.1 (Temporal Invariance)

$$\forall(a \ b : \text{TimedPulse } 2)(k : \text{Time}), \text{FA}(D_k(a), D_k(b)) = D_k(\text{FA}(a, b))$$

We can represent the circuits on each side of the equality in Citrus as shown in Listing 4.17 using the combinators defined earlier in the paper:

```
Delay-FA = ((Delay {1} k) *** (Delay {k} {1} k)) >>> FA
FA-Delay k = FA >>> Delay k
```

Listing 4.17: The two circuits used in Proposition 4.3.1.1

To prove the equivalence we proceed in three steps. The first step is to establish a coinduction hypothesis stating when the gate may be considered “done” and able to return to its initial state. The second step is to construct the sequence of inputs, keeping track of the intermediate outputs and updated circuits. The third step is to write down the proof obligations, namely, that the intermediate outputs are equal and that the final, updated circuits are bisimilar. We refer to this pattern as “setting up the equivalence” or “proof setup” and make use of it in the remaining case studies. Listing 4.18 shows the proof setup for the proof of Listing 4.17.

```
temp-inv :
-- the coinduction hypothesis
(coind : proj2 (runPG (proj2 (runPG Delay-FA (inp-1 0))) (inp-2 11))
  ≈⟨ ⟩≈
  proj2 (runPG (proj2 (runPG FA-Delay (inp-1 0))) (inp-2 11))) →
-- constructing the sequence of inputs
let (DelFA-out-1 , DelFA-next-1) = runPG Delay-FA (inp-1 0)
    (DelFA-out-2 , DelFA-next-2) = runPG DelFA-next-1 (inp-2 11)
    (FADel-out-1 , FADel-next-1) = runPG FA-Delay (inp-1 0)
    (FADel-out-2 , FADel-next-2) = runPG FADel-next-1 (inp-2 11)
-- the proof obligations
in (DelFA-out-1 ≡ FADel-out-1) ×
   (DelFA-out-2 ≡ FADel-out-2) ×
   (DelFA-next-2 ≈⟨ ⟩≈ FADel-next-2)
```

Listing 4.18: The proof setup for Proposition 4.3.1.1. It consists of three parts: assuming the coinduction hypothesis; the relevant input sequence; and the equalities and equivalences that form the proof obligations.

With this setup we can then prove the theorem (i.e., the three proof obligations given in Listing 4.18) with three corresponding equations as shown in Listing 4.19:

```
proj1 (temp-inv coind) = refl
proj1 (proj2 (temp-inv coind)) = refl
proj2 (proj2 (temp-inv coind)) = coind
```

Listing 4.19: The proof of temporal invariance consists solely of three equations which Agda validates as true.

Using Citrus with all of its provided definitions and operators, there is minimal difficulty in setting up and proving the above theorem. The only manual work it requires is to establish a given temporal path, e.g., the timestamps required for the input pulses, and the corresponding coinduction principle. It might be possible to automate this manual work as well using arithmetic solvers, but we leave it for future work.

4.3.2 Commutativity of the FA and LA Gates

The second SCE composition law states that both the First Arrival and the Last Arrival gates are each commutative, i.e., the behavior of both gates is the same regardless on which wire the input pulses arrive. Proposition 4.3.2.1 states the case only for the First Arrival gate; the statement is nearly identical in the case of the Last Arrival gate.

Proposition 4.3.2.1 (Commutativity)

$$\forall (a \ b : \text{TimedPulse } 2), \text{FA}(a, b) = \text{FA}(b, a)$$

The proof setup follows the same pattern as that of Proposition 4.3.1.1 with one exception: In this case we do not need a coinduction hypothesis as the final proof obligation is merely reflexively true. Agda is able to validate this on its own (through the use of a helper function). Listing 4.20 shows the setup code:

This proof is even simpler than the proof of temporal invariance, as we do not need a coinduction hypothesis, because the gates have identical internal state as determined by

```

FA-comm :
  let (FA-out-1 , FA-next-1) = runPG FA    (inp-1 0)
      (FA-out-2 , FA-next-2) = runPG FA    (inp-2 11)
      (FAcomm-out-1 , FAcomm-next-1) = runPG FA    (flip (inp-1 0))
      (FAcomm-out-2 , FAcomm-next-2) = runPG FAcomm-next-1 (flip (inp-2 11))
  in (FA-out-1 ≡ FAcomm-out-1) ×
      (FA-out-2 ≡ FAcomm-out-2) ×
      (FA-next-2 ≈⟨ ⟩≈ FAcomm-next-2)
proj1 (FA-comm) = refl
proj2 (proj2 (FA-comm)) = refl
proj2 (proj2 (FA-comm)) = ≈refl

```

Listing 4.20: The First Arrival Gate is commutative in the sense that its inputs can be given in any order.

Agda. Thus we can resolve the proof obligation using the helper function `≈refl` which discharges reflexively bisimilar terms (analogous to `refl` for inductive terms). The proof for Last Arrival is identical so we do not show it here.

4.3.3 Distributivity of the FA and LA Gates

The third SCE composition law states that First Arrival distributes over Last Arrival. This law is useful practically, in that it permits us to replace a circuit with four gates by a circuit with three gates. However, it is also interesting from a logical perspective, because the classical analogues of First Arrival and Last Arrival are the OR and AND gates. In Boolean logic conjunction and disjunction enjoy a distributive relationship: $(M \vee (A \wedge B)) = (A \vee M) \wedge (B \vee M)$. Here, we show that the SCE analogue to this distributive law holds as well.

This case study comes from common transformations done to classical CMOS circuits to optimize area and performance. The question is whether $\text{Circuit1} \approx \langle \rangle \approx \text{Circuit2}$ (where *Circuit1* and *Circuit2* are the top and bottom circuits in Figure 4.4, respectively); if so then a designer can optimize a circuit that uses *Circuit2* (which has more gates and a longer delay) by replacing it with *Circuit1* (which has fewer gates and a shorter delay). SCE designers strongly suspect the truth of this equivalence, but there is no proof that

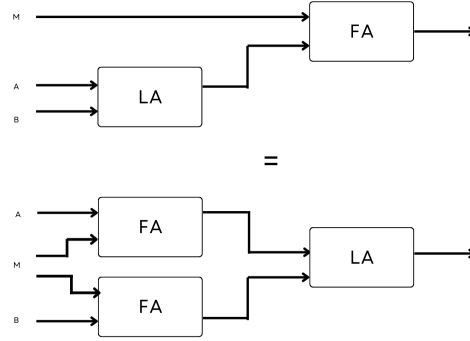


Figure 4.4: Schematic representation of the distributive law translated to pulse-based SCE gates.

We refer to the top circuit as Circuit1 and the bottom circuit as Circuit2.

```

Circ1-eq-Circ2-MAB :
  (coind : proj2
    (runPG (proj2 (runPG (proj2 (runPG Circ1 inp-M')) inp-A')) inp-B')
    ≈⟨ ⟩≈
    proj2 (runPG (proj2 (runPG (proj2 (runPG Circ2 inp-M)) inp-A)) inp-B)) →
  let (circ1-out-1 , circ1-next-1) = runPG Circ1 inp-M'
  ...
  (circ2-out-1 , circ2-next-1) = runPG Circ2 inp-M
  ...
  in (circ1-out-1 ≡ circ2-out-1) ×
  ...
  (circ1-next-3 ≈⟨ ⟩≈ circ2-next-3)
proj1 (Circ1-eq-Circ2-MAB coind) = refl
proj1 (proj2 (Circ1-eq-Circ2-MAB coind)) = refl
proj1 (proj2 (proj2 (Circ1-eq-Circ2-MAB coind))) = refl
proj2 (proj2 (proj2 (Circ1-eq-Circ2-MAB coind))) = coind

```

Listing 4.21: Distributive law proof setup. Given a sequence of inputs M, A, B there are inputs M', A', B' such that the two are behaviorally equivalent.

it is correct.

The distributive law we are trying to prove may be stated as $\text{Circuit1} \approx \langle \rangle \approx \text{Circuit2}$. The proof setup is shown in Listing 4.21. In order to decompose the problem, we will investigate it by cases. As described earlier the SCE gates work in a cyclical fashion, returning to an initial configuration after receiving a pulse on every wire. This behavior means that there are eight cases to consider, one for each possible ordering of pulses arriving on the A , B , and M wires. For example, in one case we examine the input sequence in which the M wire pulses, then the A wire followed by the B wire. We prove that the outputs of the two circuits Circuit1 and Circuit2 for this input sequence are equal. At this point, the gates should each be back to their respective initial configurations—except that they are not exactly. In an untimed setting this would be true, but in a timed setting they have each updated their clocks and so are technically not the same as the original configurations (though they are the same up to that timing information). To overcome this problem we employ a standard coinductive technique analogous to strengthening an inductive hypothesis: we assume a coinductive hypothesis stating that the new configurations are time-abstract bisimilar. Essentially, this hypothesis states that we went through one step of testing that the circuits yield the same behavior for the same input sequence, and that the remaining infinite steps will look exactly the same—thus if the first step worked (which it did) then the remaining steps must work as well.

4.3.4 Bitonic Correctness

Our final case study is on bitonic sorters, a real-world SCE circuit that was also used as a case study in the Pylse paper [1]. A bitonic sorter is a parallel sorting network composed of min-max blocks. An n -width bitonic sorter implementation is correct if it

pulses its n th output wire on the n th input to arrive. PyLSE is unable to prove the correctness of a particular 4-bit SCE bitonic sorter implementation via model-checking, because the compiled timed automaton to be model checked contained too many states to be efficiently validated. However, with Citrus its proof is only a few lines long. We show the Citrus program implementing the bitonic sorter in Listing 4.22 and its proof of correctness in Listing 4.23.

```

bitonic-sorter-4 : PulseGate (TimedPulse 4) (TimedPulse 4)
bitonic-sorter-4 = (min-max *** min-max) >>>
                  ((arr perm1) >>>
                   ((min-max *** min-max) >>>
                    ((arr perm2) >>>
                     ((min-max *** min-max))))))

```

Listing 4.22: The definition of a 4-bit bitonic sorter. The *perm1* and *perm2* functions are wire management arrows.

```

bitonic-correct :
(coind : proj2 (runPG (proj2 (runPG (proj2 (runPG
(proj2 (runPG bitonic-sorter-4 test-pulse-a)) test-pulse-b))
test-pulse-c)) test-pulse-d) ≈( )≈ bitonic-sorter-4)
  let (bit4-out-1 , bit4-next-1) = runPG bitonic-sorter-4
    (test-pulse-a)
    (bit4-out-2 , bit4-next-2) = runPG bit4-next-1
    (test-pulse-b)
    (bit4-out-3 , bit4-next-3) = runPG bit4-next-2
    (test-pulse-c)
    (bit4-out-4 , bit4-next-4) = runPG bit4-next-3
    (test-pulse-d)
  in (proj1 bit4-out-1) ≡ (* :: ∅ :: ∅ :: ∅ :: ∅ :: []) ×
    (proj1 bit4-out-2) ≡ (∅ :: * :: ∅ :: ∅ :: ∅ :: []) ×
    (proj1 bit4-out-3) ≡ (∅ :: ∅ :: * :: ∅ :: ∅ :: []) ×
    (proj1 bit4-out-4) ≡ (∅ :: ∅ :: ∅ :: * :: ∅ :: []) ×
    bit4-next-4 ≈( )≈ bitonic-sorter-4
proj1 (bitonic-correct coind) = refl
proj1 (proj2 (bitonic-correct coind)) = refl
proj1 (proj2 (proj2 (bitonic-correct coind))) = refl
proj1 (proj2 (proj2 (proj2 (bitonic-correct coind)))) = refl
proj2 (proj2 (proj2 (proj2 (bitonic-correct coind)))) = coind

```

Listing 4.23: The 4-wide bitonic sorter correctness proof.

The proof in Listing 4.23 is establishing the four proof obligations (in addition to the fifth

obligation establishing bisimilarity), i.e., that after running all input pulses, the sequence of output pulses should come in order, from first to last wire of the bitonic sorter itself. Given the machinery provided by Citrus, the proofs are straightforward. We further note that the PyLSE paper attempted the 8-bit bitonic sorter as well but similarly it succumbed to state explosion and was not able to be model checked. We can see from the above code that extension to the 8-bit case is trivial and correctness can be proven in an identical manner.

4.3.5 Discussion

Citrus as a language greatly expands the capabilities for reasoning about timed, reactive behavior, specifically in the case of SCE. It drastically reduces the amount of state-based checking that is required in PyLSE-TA and generalizes properties to a semantic level as opposed to singular instance-based. However, its proof principle is still relatively manual, and nevertheless tied to a sort of exploration of the design space. In particular, it is still difficult to reason generally about timing properties. One way to fix this is to focus on the mathematical nature of time as opposed to the, up to now, discretized nature that is necessary for model checking and type checking. In some sense, we would like to simply reason about timed functions in the fashion of mathematicians or physicists, discretizing at will or when necessary. This will additionally allow us to consider timed, reactive systems in a more general sense, without the specific mooring of superconducting circuits.

Chapter 5

Topology of Timed, Concurrent Systems

In this chapter we present, PLATA, our third and final language. PLATA is a DSL and library, much like Citrus, written in the Lean theorem prover. It builds on Citrus (and PyLSE-TA) in two ways. First, it provides a semantics for timed, concurrent systems that generalizes individual traces to sets of traces that are parameterized by timing information. Second, it builds a number of combinators and lemmas which allow the user to incrementally build systems via specifications. PyLSE-TA is able to consider all possible executions of a specific, manually determined, input trace. As noted, this makes it difficult to generalize beyond specific input sequences as well as suffers from compositionality problems. Citrus improved on PyLSE-TA by making compositionality much easier, while also defining equivalence over elements. Nevertheless, it was still beholden to manually determined traces.

PLATA generalizes beyond traces to *sets of* traces. For instance, for a given time constraint, e.g. “the traffic light changes from green to red within 2 milliseconds”, there are multiple satisfying traces. In the example, we have those that transition from green

to red in 1.5 milliseconds, 1 millisecond, 2 femtoseconds, etc. This gives us a set of traces. Moreover, this set of traces can be intersected, unioned, and complemented, as any field of sets is automatically a Boolean algebra. Nevertheless, we need to keep track of multiple timing constraints, which are by default defined over the Real numbers. A natural tool for keeping track of this information is topology. Thus, PLATA commits to a semantics of timed, concurrent behaviors defined as Real-valued traces and is able to leverage topology to do so.

The switch to a much more mathematical semantics would naturally push us away from a programmatic language design into a more proof-oriented design. Indeed, PLATA is written in Lean which allows it to take advantage of a large library of theorems from general topology. Our goal however is to write timed, concurrent systems like functional programs, and to do this, we situate PLATA in the framework of the Refinement Calculus (RC). The Refinement Calculus is a formal system in which the user writes preconditions and postconditions as predicates and links them together by transformers, or functions that preserve certain continuity assumptions. In fact, the RC can be thought of as a formal semantics for Dijkstra’s predicate transformers and Guarded Command Language [41]. The fundamental notion, beyond predicate transformers, is that of refinement, in which one transformer can be seen as more descriptive than another. If one predicate transformer refines another, we see it as more fully defined and closer to executable code. Refinement forms a preorder and the RC proves several lemmas about refinement which allow a user to begin with a broad pair of pre and postconditions, and then step-wise apply refinement lemmas to obtain a more concrete version. We detail the topological semantics of PLATA, show how this can be adapted to the Refinement Calculus, then show it can be used to build and verify simple, timed concurrent systems.

5.1 The Finitely Live Topology for Timed Automata

One of our objectives is to allow algebraic reasoning about timed events over networks of concurrent objects. Since TA already model such systems, it is worth investigating whether we can give TA a topological semantics that allows for such reasoning. The answer is a qualified “yes”—technically we can define such a semantics, but the result does not meet all of our requirements. In this section we explain how TA can be given a topological semantics as an extension of the pointful Finitely Live topology for LTS and why it is not suitable as a foundation for PLATA (but can still potentially be useful in some ways).

5.1.1 Topologizing TA Directly

In Chapter 2 we reviewed an existing method for describing Labeled Transition Systems (LTS) topologically. We can view a TA as an LTS extended with edge guards on $\mathbb{R}$ -valued clocks, and extend the Finitely Live topological description of LTS to a pointful Finitely Live-like topological description of TA. In this extension, the points are TA traces (rather than LTS traces) and the opens are subsets of points, just as for LTS. Let us call the result the *Continuously Live* topology, which is the same as the Finitely Live topology but with the $\mathbb{N}$ index replaced by $\mathbb{R}$. However, we immediately run into a fundamental problem: because the clocks are $\mathbb{R}$ -valued, trace sets contain an uncountably infinite number of traces. For example, consider the TA given in Figure 2.3. Between the start state $\emptyset$ and the next state $\langle q_1, x := 2.1 \rangle$, there are possible delays of 2.11, 2.111, 2.111..., 3.14..., etc; there are, in fact, traces for each possible valuation of the clock that matches the edge constraint.

The Finitely Live topology is suitable for LTS but not so much for TA because ultimately the Finitely Live topology only describes *discrete* traces, which makes it possible

to enumerate or iterate over sets of traces. Viewed from the perspective of programming language semantics, we see this ability as a hallmark trait of *domains* [47], i.e., the ability to iterate or recurse over objects. A domain is a particular kind of partially ordered set (with a lot of extra structure) at the foundation of denotational semantics. Domains are most appropriately seen as structures in order theory, but they have clear analogues in both pointful and point-free topology in the form of the Scott topology and spectral, algebraic locales [24]. We bring this information up because it points at the crux of the problem for extending the Finitely Live topology to cover TA via the Continuously Live topology: the fact that the result does not give us the same “domain-like” ability. The $\mathbb{R}$ numbers are not a good fit for domains (though there is work on this subject [48]) and if we want to apply the Finitely Live topology in a more useful way we will need to work over an abstraction of TA rather than directly over TA themselves.

5.1.2 Topology of Discretized TA

Discretizing TA is a well-known technique that is required in order to apply model checking to TA. The trick is to coarsen the state space so as to remove uncountability. Coarsening the state space traditionally comes in the form of applying an equivalence relation over the state space, and the most common equivalence relations used for TA when model checking are the region and zone constructions [5], as discussed in Chapter 2. The region construction results in a finite set of equivalence classes. The zone construction is possibly infinite, but in practice is further discretized to a finite set of equivalence classes as well. The intuition behind both constructions is that traces which can delay for roughly similar amounts of time, and thus move more or less together, are grouped together and mapped onto a single temporal interval. Performing this discretization allows the (uncountably infinite) TA trace semantics to be treated in a domain-like fashion. As an

example, consider Figure 5.2, taken from [49], and note that the $\mathbb{R}$ -valued clock labels on the TA have disappeared in the corresponding zone graph, leaving us with a countable number of possible traces of the zone graph (as opposed to the TA itself).

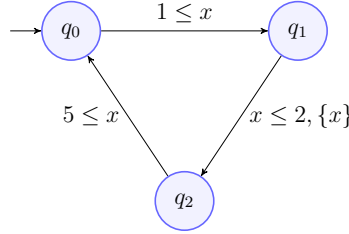


Figure 5.1: A timed automaton. Image reproduced from [49].

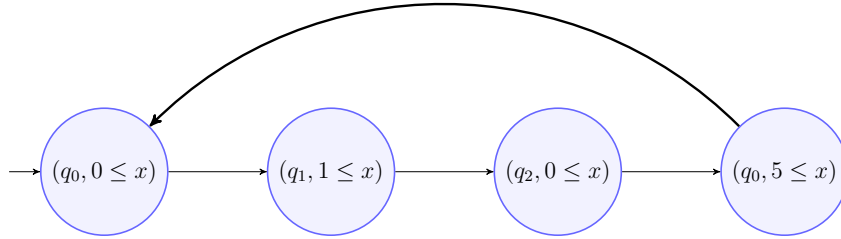


Figure 5.2: The zone graph corresponding to the TA in Figure 5.1. Image reproduced from [49].

The key intuition for the purposes of topologizing TA is that a discretized TA is essentially an LTS, and thus we can apply the Finitely Live topology to describe its semantics. We assume some equivalence relation on traces (e.g., as induced by regions or zones) and apply it to the Continuously Live trace sets to quotient the points, yielding countable sets of equivalence classes on traces. We will call the result the Quotiented Continuously Live topology, or *QCL* topology for short.

We believe that this construction is useful for its own sake and provide an argument below, though we leave a thorough investigation to future work in favor of a completely different topological approach in PLATA.

Model checking by discretization is currently the main mechanism for reasoning about TA. While the use of regions and zones opens the door to iteration-based techniques, we

observe that existing techniques do not take advantage of the algebraic properties of the equivalence relation. Despite the fact that model checkers are based on this equivalence relation approach, they do not expose the objects themselves. That is, for the working researcher, it could be interesting to reason about equivalence classes of traces in a semi-manual (semi-automated) fashion. Our argument is that for a programming language researcher, using a model checker is a monolithic approach that obscures internal, intermediate reasoning steps. By expressing the discretization equivalence relation explicitly in a topological setting we can provide a much richer set of tools for reasoning-based problems and tasks.

While our support for this viewpoint is still speculative, we offer the following example of how taking a topological viewpoint (where we see the domain as a kind of Scott topology) can be used for traditional tasks in TA and model checking.

Example 1 (An Algebra of Regions)

Consider the following two traces:

$$\begin{aligned} \text{trace}_1 &= \emptyset \xrightarrow{2.1} \langle q_1, x := 2.1 \rangle \xrightarrow{6.0} \langle q_1, x := 8.1 \rangle \xrightarrow{\text{push}!} \langle q_2, x := 8.1 \rangle \xrightarrow{1.9} \langle \dots \\ \text{trace}_2 &= \emptyset \xrightarrow{2.56} \langle q_1, x := 2.56 \rangle \xrightarrow{6.09} \langle q_1, x := 8.65 \rangle \xrightarrow{\text{push}!} \langle q_2, x := 8.65 \rangle \xrightarrow{1.9} \langle \dots \end{aligned}$$

We expect these two traces to live in the same equivalence class E . We can thus reason about them by thinking not of a set of traces, but by one representative element. Lean has ample support for quotient types and thus we can reason about representative elements natively and with ease. This representative element is a trace, or rather a function $\mathbb{R} \rightarrow \Sigma$ with Σ the state space. We can assume we have put some sort of topology on the state space, for instance the product topology over some discrete topologies (representing the finite set of clocks, etc.). As a $\mathbb{R}$ -valued function we can talk about its topological properties. In this case, we might ask what the function converges to. We can set up convergence cleanly in Lean using neighborhood filters [50] and deduce that traces from

this equivalence class will always converge to $\langle q_2, x \geq 10 \rangle$.

This example demonstrates that traditional model checking queries can be reformulated into topological versions with the same character, e.g., turning a model check about state reachability to a question of convergence of a sequence. This is a characterization we have not seen in prior literature and which we think could be a powerful tool in the programming language researcher’s toolbox.

5.1.3 Discussion of TA Topology

While the Quotiented Continuously Live (QCL) topological semantics for discretized TA does allow for equational reasoning and may be useful, there are two main shortcomings that lead us to abandon it for use in PLATA in favor of a different topological approach. One shortcoming stems from the fact that the Continuously Live topology is pointful, as opposed to point-free: it requires reasoning over equivalence classes of $\mathbb{R}$ -valued traces, which can make reasoning difficult. The other shortcoming is that the traces themselves reference the graphical structure of the TA, and thus this approach inherently has the same issue discussed in the introduction: the lack of modularity between a TA model and a property encoding. The topological semantics that we propose in Section 5.2 has neither shortcoming.

5.2 A Point-Free Topological Semantics for PLATA

For PLATA’s semantics we move to a point-free topology, i.e., we abstract away any notion of points (e.g., traces) and operate solely in terms of an abstract locale. The basis of the semantics is a *clock locale* that allows us to reason about time. We construct the locale in terms of zones, where zones are convex polyhedra describing a range of clock

values. Zones are the same abstraction used to discretize TA; however, the way that we use them is different from the way that we used them in the (pointful) QCL topology. Recall that for the QCL topology we use zones to quotient the traces, partitioning the uncountably infinite number of traces into a countable number of equivalence classes. The QCL opens are sets of trace equivalence classes, not zones per se—zones do not appear in the semantics, they are only used to define the equivalence relation. The quotient construction itself is not the problem, it is that the construction involves $\mathbb{R}$ -valued traces and inherently incorporates the underlying graph structure of an automaton. The PLATA development repurposes the quotient construction to build a similar topological semantics without reference to traces (and therefore without reference to an automaton graph structure).

The clock locale lets us reason about time (in terms of zones), but we also need to be able to reason about events. The PLATA semantics is parameterized by an event locale over some set of relevant events. The product of the clock locale and the event locale is necessarily also a locale, and this is the specific locale that we would reason over for a particular PLATA model. This point-free topology allows for the same sort of algebraic flavor as the QCL topology, but has a structure that is more amenable to reasoning (in that we’re reasoning over pairs of zones and events rather than equivalence classes of $\mathbb{R}$ -valued traces) and that is independent of any automaton or graphical structure. We expand on the details of the PLATA topological semantics in the remainder of this section.

5.2.1 Defining the Clock Locale

We recall the definition of a locale: a complete lattice (i.e., that has least upper bounds and greatest lower bounds for all subsets) that satisfies the following distributive

law (where $\sqcup$ and $\sqcap$ are the least upper bound and greatest lower bound, respectively):

$$\forall x \in S, x \sqcap \bigsqcup S = \bigsqcup \{x \sqcap s \mid s \in S\}$$

We want the elements of our locale to be zones, and our job is to define a locale of zones. We will proceed by proving that zones form a partial order, and then extend that partial order to a locale. However, first we must discuss how to represent a zone.

A zone can be viewed in two perspectives, as an abstract collection of values constituting a convex polyhedron (in our use-case the values would be points in time for each clock), or concretely as a data structure representing the points in the convex polyhedron. The first view can be a bit too abstract to work with. Conversely, the data structure viewpoint is more concrete but still sufficiently general to build our locales. We will adopt the same data structure from the model checking community to represent convex polyhedra, namely the difference bound matrix (DBM). To describe DBMs we begin with a more formal treatment of zones.

A zone can be presented as a formula of one the following forms: $x - y < k$ or $x - y \leq k$, where $k \in E\mathbb{Z}$ is an element of the extended integers (the regular integers affixed with a top and bottom element), x and y are clocks, and both strict and weak less-than are valid comparison operators. We also allow conjunctions of formulas, so if z_1 and z_2 are zones then so is $z_1 \wedge z_2$. A zone is therefore a system of linear inequalities and describes a necessarily convex set. Zones describe *clock valuations*, which are maps from the set of clocks to $E\mathbb{Z}$; every zone is mapped to the set of all clock valuations that validate the formula, i.e., for which the clock values are solutions to the zone. We can flip between two equivalent viewpoints on zones as convenient: that of a zone as an abstract convex set, and a slightly more concrete representation of a zone as a set of linear inequalities of clock constraints.

For n clocks, the convex set of valuations is determined solely by pairwise comparisons

[51]. That is, for a set C of clocks, $(1 + |C|)^2$ values are sufficient to describe the resulting solution set. The additional 1 in the formula is a reference clock, or zero clock. We define a DBM as a square matrix of dimension $(1 + |C|) \times (1 + |C|)$ in which the cells are pairs of a comparison operator and a time value draw from $E\mathbb{Z}$. The cell at position (i, j) refers to the i th clock minus the j th clock, compared using the cell's first component to the cell's second component. The example below is reproduced from [5] and corresponds to the zone $z - 0 < 0 \wedge y - 0 \leq 20 \wedge y - x \leq 10 \wedge x - y \leq -10 \wedge 0 - z < 5$.

$$\begin{pmatrix} (0, \leq) & (0, \leq) & (0, \leq) & (0, \leq) \\ (20, <) & (0, \leq) & (-0, \leq) & \infty \\ (20, \leq) & (10, \leq) & (0, \leq) & \infty \\ \infty & \infty & \infty & (0, \leq) \end{pmatrix}$$

In the remainder of this section we will use the term “zone”, but our Lean library uses both clock valuations as well as DBMs. A DBM may potentially represent a non-convex space if the set of constraints, read as a system of linear inequalities, does not have a solution. Currently a PLATA user has to manually ensure convexity, but it is straightforward to add automated convexity checks.¹ In the model checking community, convexity may be checked by treating a DBM as a graph and looking for negative cycles between clocks.

Proposition 5.2.1.1 (Zones Form a Partial Order)

Using the definition of zones as DBMs, we are able to use the partial order given by Dill [51], formalizing and proving it correct in PLATA. The order is first defined on DBM cells and then lifted pointwise to entire DBMs. For two cells (i.e., pairs of comparison operator and value from $E\mathbb{Z}$), $(cmp_1, ez_1) \leq (cmp_2, ez_2)$ if $ez_1 < ez_2$ or if $ez_1 = ez_2$ and $cmp_1 \leq cmp_2$. For comparison operators, we take $\leq$ to be less than or equal to $<$. This is the lexicographic order on cells, which is then lifted pointwise to DBMs.

¹Doing so is part of ongoing work.

As a corollary, we note also that sets of clock valuations additionally form a partial order under usual set containment.

Leveraging Free Locales. We now have a partial order of zones, but must extend that partial order to a locale. We will begin by taking the most obvious route to building a locale, i.e., free locales, taking advantage of the known fact that the lower sets of a partial order form a locale [52]. We remark that the presentation in Tosun is based on constructive logic and formalized in Agda. We work in a traditional logic, formalizing and proving the result correct in Lean as part of the PLATA library. We then explain why this construction is not exactly suitable for our purposes, and tweak the construction to make it suit. To start, we recall one more definition, that of lower sets (also called lower closed sets or downsets):

Definition 12 (Lower Sets)

For an element x in a partially ordered set P , the lower set of x is the set of all elements in P which are less than x .

Proving that the lower sets form a locale involves establishing what operations on lower sets constitute the meet, join, supremum, infimum, top, and bottom elements, as well as a host of relations that must hold between them, such as the distributive law mentioned earlier. For zones we can define the (finite) meet of any two lower sets to be the intersection of the lower sets, and dually for join. The infimum is taken to be iterated finite meet, and dually for supremum. The top and bottom elements are the entire set and empty set respectively. As described in [53] this definition constructs the *free* locale with the elements of the partial order as its generators. The free locale is the locale in which only the equations of the structure hold; it can be thought of as the “most general” locale.

We can instantiate the above theorem on clock valuations and obtain a locale. This

construction is generic and unrelated to any underlying graph or transition structure. There are two main problems with the result: (1) the locale is over sets of zones rather than being over zones; and (2) it does not discriminate between convex and non-convex zones. The union of two convex sets is not necessarily convex, and thus the join of two sets of clock valuations (DBMs) may not result in a valid (i.e., well-defined) zone. Therefore the free locale, while capturing the general features of reasoning about timing and clocks, is not suitable for reasoning about timing and clocks that must abide by linear inequality constraints.

Fixing the Free Locale.

We address the above issues in the follow sequence of steps. The first problem is that we obtained a collection of lower sets, as opposed to zones which is what we would like our formal opens to be. We need a way to collapse these sets of lower sets onto a single lower set, which we imagine as the total union of the collection. Pleasantly, lower sets have a well defined supremum relation, taking a set of lower sets to a lower set. Thus for some set S_L of lower sets we have that $\bigsqcup S_L$ maps to a single lower set L , which is a collection of zones. We can imagine a zone as a (convex) set of clock valuations or as the more concrete DBM. For conceptual simplicity, we use the former. Thus, L contains a *set* of a set of valuations. We therefore would still like to union these, obtaining one large collection of clock valuations, defining a single abstract region of space. It also happens that the algebraic structure consisting of nested sets has a supremum operator which does indeed collapse a set of sets down to a single set. Performing this we have obtained a large region of space based on clock locales.

The second major problem is that despite being formed from convex sets, the union of convex sets is not necessarily convex. We solve this by overapproximating the region then taking the convex hull of these points. This results in a set of clock valuations

representing our enlarged region. We would like to finish here, mapping a supremum of lower sets to a single convex hull, but to use a quotient construction, we require a binary relation on the free locale. To this end, we wrap the convex hull in a singleton set and take the lower closure, resulting in a single lower set. Interpreting the equation as a relation on the product of the free locale with itself, we then generate its congruence closure and take this as our quotient. The relation in full is presented in Definition Lower Sets 13. We adopt the style of a “presentation” where the left hand side of the vertical bar is read as a set of generators, in our case, the lower sets, which creates a free locale, and the right hand side presents an equational relation by which to quotient the free locale.

Definition 13 (Clock Locale)

For a set S of lower sets of sets of clock valuations,

$$\langle \downarrow \text{Set (Clock Valuations)} \mid \bigsqcup S = \downarrow \{ \text{convex-hull}(\bigsqcup_S (\bigsqcup_L S)) \} \rangle$$

where $\downarrow T$ is the lower closure of a set T , generating all sets contained in T and the subscripts L and S respectively refer to supremum of lower sets and nested sets.

Ultimately, we squash all of the elements in the free locale into convex hulls, giving a point-free topological space of convex objects that may be manipulated algebraically. We note that every congruence class has a canonical representative, namely the supremum of all its elements. In this way, the congruence class of a union of an arbitrary number of lower sets is mapped to the lower set of a singleton, and as a congruence class its supremum is the canonical representative. Moreover, the supremum of a lower set of a singleton is the singleton. In this way we can reason directly about the convex hulls while still enjoying the point-free topological structure. Note further that taking convex hulls of clock valuations requires the set of clock valuations to be at least an additive

commutative monoid. Dill [51] references DBMs as being Kleene algebras and thus we axiomatize the additive commutative monoid instance in the PLATA development. It is straightforward to prove this fact and it is son-to-be-completed work to unify the treatment of zones, clock valuations, DBMs, and formal opens as interderivable from one abstract structure.

The Clock Locale. In summary, to create the clock locale we generate the free locale over the set of clock valuations and then quotient by the convex hull relation. The result of this construction is a locale over zones (the “clock locale”) that allows us to reason about time using point-free topology.

5.2.2 Incorporating an Event Locale

The clock locale is only suited to reasoning about clocks, while we want to specify behaviors that are also parameterized by sets of events or other collections of data. In addition to being well behaved under quotients, locales also preserve products, i.e., the product of two locales is a locale. PLATA allows the user to create richer and more complicated specification structures, e.g., incorporating arbitrary domains of events, as long as those structures also form locales.

Given an event locale, incorporating it into our semantics is as simple as forming the product of the clock locale (which is always the same, for all models) and the event locale (that is specific to a given model). We will call an instance of this product locale a *PLATA locale*. A PLATA locale is necessarily a locale by construction, and all of our results and reasoning framework apply to it unchanged.

5.2.3 The Relation Between PLATA and the QCL Topology

The PLATA semantics (i.e., the PLATA locale as described above) is sufficient by itself as a topological semantics of timed events and, as we show in Section 5.3, allows for using the Refinement Calculus to build PLATA programs. However, it is interesting to investigate the relation between the PLATA semantics and the QCL topology that we constructed for discretized TA. We believe that we can enrich the PLATA point-free topology with points to recover the QCL pointful topology, providing a kind of “soundness” theorem for the PLATA topology—that is, the PLATA semantics abstracts, in a way, the more concrete pointful semantics of TA. Proving this relation formally is ongoing work, but should be straightforward. To show this relation between PLATA and the QCL topology we use the notion of a *topological system* [24] (note the distinction between a topological *space* and a topological *system*):

Definition 14 (Topological System)

A topological system is a tuple $\langle X, \models, A \rangle$ where X is any set of points, A is a locale, and $\models$ (pronounced “models”) is a binary relation between X and A satisfying the following laws:

- Let S be a finite subset of A then, $x \models \bigcap S \iff \forall a \in S, x \models a$
- Let S be an arbitrary subset of A , then $x \models \bigcup S \iff \forall a \in S, x \models a$

The idea is that the points give a more concrete description of the locale and to do that, they must simply preserve the behavior of arbitrary joins and finite meets.

A topological system connects a topological space (i.e., a carrier set X with its set of opens) and an abstract locale, via the $\models$ relation, by ensuring that there exists a way to partition the concrete points of the topological space into the opens of the abstract locale. In other words, it gives a way to reconstruct a concrete topological space from an

abstract locale. To show the relation between PLATA and the QCL topology, we must show that there exists a topological system that relates the two. Specifically we must show a relation between the clock locale and the QCL topology; having shown that, the relation with a PLATA locale follows trivially.

Proposition 5.2.3.1 (Clock Locale-QCL Topological System)

Proof in progress.

This proposition establishes a result analogous to that of the soundness and completeness of timed automata with their zone graphs. If the QCL topology models TA then the clock locale models zone graphs, and moreover the two simulate each other modulo their respective equivalence relations.

5.2.4 Converting Between PLATA and TA

We briefly explore the question of whether PLATA models and TA models are interconvertible. That is, since TA plus model checking and PLATA are complementary verification approaches, can we move between the two at will depending on the task at hand? We do not have any formal results on this question, leaving it to future work, but we provide some thoughts.

From TA to PLATA. It is well-known how to discretize a TA into a zone graph. We have already shown how to then construct the corresponding QCL topology (and the procedure is encoded in the PLATA library). As discussed in Section 5.1.3, this topology already makes algebraic reasoning possible, but its pointful nature can be a hindrance. The open question in this direction is whether we can abstract the QCL topology into a corresponding PLATA locale. We believe that the answer is “yes”, but have not developed such a construction yet.

From PLATA to TA. Section 5.2.3 conjectures (awaiting a complete formal proof) that

we can enrich a PLATA locale to the QCL topology. The open question in this direction is whether we can extract a zone graph (which describes a TA) from a QCL topology. Again we believe that the answer is “yes”, but have not developed such a construction yet. We take inspiration from the work in the process algebra mCRL2, which offers a large tool suite for analyzing concurrent systems according to systems of recursive equations [54]. As described in the work of Bunte [55], there is a tool which takes a set of process algebra equations and performs state space generation, effectively generating a labeled transition system from a process equation. We imagine that there is a way to encode PLATA specifications as process algebra equations and then leverage the tool directly, or alternatively adapt the same state space generation technique to PLATA specifications.

5.3 The Refinement Calculus for PLATA

We have provided a topological setting for reasoning over timed events, but we have not yet provided any interface that a programmer could use to actually create a model in this setting. We would like PLATA locales to be used in a programmatic way to specify and build programs over timed events. We turn to the Refinement Calculus for this purpose.

Back and von Wright [21] observe that complete lattices are not only sufficient to interpret the Weakest Precondition Calculus [41], but are also sufficient to interpret a notion of refinement, yielding the Refinement Calculus. To instantiate the Refinement Calculus for PLATA we must show that (1) our domains (that we are writing predicates over) are complete lattices, and (2) that monotone maps between domains (i.e., predicate transformers) also form a complete lattice.

These requirements are met naturally by our topological semantics. In the first case, locales are complete lattices with an additional distributive law. This means that we are

free to construct predicates over zones and be confident they will behave analogously to the arbitrary predicates used in the Refinement Calculus. In the second case we observe that the Refinement Calculus transformers are order preserving (monotonic) functions over a complete lattice, and therefore complete lattice homomorphisms. In this respect, the proof is complete because every complete homomorphism is a frame homomorphism². That is, a complete lattice homomorphism preserves arbitrary joins *and* arbitrary meets, whereas a frame homomorphism preserves arbitrary joins and only *finite* meets. We can thus be confident that using the Refinement Calculus transformers over our PLATA Locale predicates is well-behaved.

With these requirements in place we can describe more concretely what the PLATA locale instantiation of the Refinement Calculus looks like. Recall that an element of a PLATA locale is a pair of a zone and an event. Maps over a PLATA locale transform (zone, event) pair into new (zone, event) pairs; they are “timed event transformers”, or more briefly “PLATA transformers”. A predicate over a PLATA locale is a description of the set of (zone, event) pairs that satisfy some condition. A PLATA transformer is a map between such predicates.

By leveraging the Refinement Calculus we are able to specify a system model as a pair of pre- and post-condition predicates over the (zone, event) pairs, and then apply PLATA transformers to successively refine a model using the Refinement Calculus operators and lemmas into a concrete program that satisfies the pre- and post-conditions by construction. The Refinement Calculus notions of refinement, assertion, update, assumption, and sequential composition of specifications, as well as all other aspects of the Refinement Calculus, all carry over immediately to PLATA locales and transformers.

While the set of all monotone maps over a PLATA locale forms a complete lattice, some of those maps are useful and some are not. As we previously described in Chapter 2,

²a frame homomorphism is a contravariant locale homomorphism

the following four functions can be used to create useful PLATA transformers: *assertion*, *assumption*, *update*, and *sequential composition* (in the definitions below, think of $\cup$ as disjunction, $\cap$ as conjunction, and $\cdot^c$ as negation):

Assertion. Written as $\{p\}$, assertion turns a predicate p into a PLATA transformer representing a precondition as $\lambda q. p \cap q$ for any predicate q . That is, $\{p\}$ describes those predicates that also satisfy p (i.e., those whose conjunction with p is not false).

Assumption. Written as $[p]$, assumption turns a predicate p into a PLATA transformer representing a postcondition as $\lambda q. p^c \cup q$ for any predicate q (where $\cdot^c$ stands for complement). That is, $[p]$ describes those predicates that are implied by p (think of $p^c \cup q$ as $\neg p \vee q$, i.e., $p \implies q$).

Update. Written as $\langle f \rangle$, update turns a function on a PLATA locale f into a PLATA transformer by calculating its set-based inverse. For a function $f : D \rightarrow D$ the transformer resulting from $\langle f \rangle$ is $\lambda q. \{x \mid f x \in q\}$. That is, $\langle f \rangle$ creates a transformer that, given a predicate q , results in a predicate p s.t. applying f to a (zone, event) in p would result in a (zone, event) in q .

Sequential Composition. Written as $f; g$, sequential composition of PLATA transformers means to apply f and then g and has the natural definition of function composition, i.e., $f \circ g$, since transformers are just functions.

We can think of a PLATA program as any term written using the refinement operators above, predicates over events and formal clock opens (which can be concrete zones), and any functions over the PLATA locale which are additionally frame homomorphisms. We now give some simple examples of system specifications and refinements using the operators we have described. For these examples we assume an event locale whose elements are sets of (unspecified) events $Events = \{\mathbf{event1}, \mathbf{event2}, \dots\}$ and a set of clocks $Clocks$

$= \{\text{clock1}, \text{clock2}, \dots\}$ (using seconds as the time units) and use DBMs (as described in Section 5.2.1) for zones, i.e., a predicate is a set of pairs of formal $(event, dbm)$ pairs.

We define the following predicates:

- $A = \{(e, d) \mid e = \{\text{event1}\} \wedge d_{1,0} \geq 0.5 \wedge d_{1,0} \leq 1.0\}$
- $B = \{(e, d) \mid e = \{\text{event2}\} \wedge d_{1,0} \geq 0.5 \wedge d_{1,0} \leq 2.0\}$
- $R = \{(e, d) \mid e = \{\} \wedge d_{1,0} = 0.0\}$

Then we can specify a system model as:

$$\{R\}; [A \cap B]$$

This specification says that we begin with no events having happened and **clock1** at 0.0, then **event1** must happen between 0.5 and 1 seconds and **event2** must happen between 1 and 2 seconds. Note that the times that **event1** and **event2** can happen overlap, but according to the predicates when **event1** happens **event2** must not be present and when **event2** happens **event1** must not be present.

We define three functions on the PLATA locale that are intended to implement this specification as follows, where **dbm_update** is a function that takes a DBM, a clock to update, and a specified value, and returns a new DBM valuation where the clock constraints reflect the updated value (the algorithm is presented in the UDBM library [56]):

- $\text{ev1} = \lambda(e, d). \text{if } e = \{\} \text{ and } 0.6 \leq d_{1,0} \leq 0.9 \text{ then } (\{\text{event1}\}, d)$
- $\text{ev2} = \lambda(e, d). \text{if } e = \{\text{event1}\} \text{ and } 1.5 \leq d_{1,0} \leq 1.9 \text{ then } (\{\text{event2}\}, d)$
- $\text{reset} = \lambda(e, d). (\{\}, \text{dbm_update}(d, 1, 0))$

We can now apply the Refinement Calculus to refine the initial specification into a concrete program:

$$\begin{aligned}
\{R\}; [A \cap B] &= \{R\}; [A]; [B] && := \text{by [21]} \\
_ \sqsubseteq \langle \mathbf{reset} \rangle; [A]; [B] &&& := \text{by } \langle \text{proof obligation 1} \rangle \\
_ \sqsubseteq \langle \mathbf{reset} \rangle; \langle \mathbf{ev1} \rangle; [B] &&& := \text{by } \langle \text{proof obligation 2} \rangle \\
_ \sqsubseteq \langle \mathbf{reset} \rangle; \langle \mathbf{ev1} \rangle; \langle \mathbf{ev2} \rangle &&& := \text{by } \langle \text{proof obligation 3} \rangle \\
_ &= \langle \mathbf{reset}; \mathbf{ev1}; \mathbf{ev2} \rangle && := \text{by [21]}
\end{aligned}$$

After discharging the proof obligations using the definition of the refines relation $\sqsubseteq$ (given in Chapter 2), this process tells us that the program $\langle \mathbf{reset}; \mathbf{ev1}; \mathbf{ev2} \rangle$, consisting of a composition of functions over the sytem’s PLATA locale, is a provably correct implementation of the original system specification.

5.4 Lean Library Formalization

In this section we describe the definitions for our PLATA formalization in Lean, as a guide to the PLATA library code that accompanies this paper. These definitions include zones, locales, and the Refinement Calculus. The Lean library code establishes the formalization of the Clock Locale topology, the language of the Refinement Calculus based on predicate transformers over complete lattices. Note that at present, the connection between the Clock Locale and both sets of clock valuations and DBMs is axiomatized, so that the refinement library is more or less standalone. This will change in ongoing work that unifies the data structures involved. The library’s defined types not only prevent so-called “user errors”, such as relating two different models which do not share the same set of actions, but also much deeper mathematical errors, such as trying to compose

two predicate transformers one of which is not strictly continuous. Overall, the library does the work of ensuring the user can reason easily and freely about specifications and refinements while also enforcing that reasoning stays within the frontiers where such reasoning is valid.

5.4.1 Locales

Locales as an algebraic structure are present in Mathlib as `Order.Frame` and we can use this definition out of the box. Locales are well-behaved objects and enjoy many usual constructions, such as taking products and coproducts. We contribute the formalization and proofs of two theorems found in [23] and [52] which are necessary to construct the locale topology, shown in Figure 5.3. We use the `LowerSet` construction, also found in Mathlib, to form the collection of downwards closed sets of a type. By instantiating the type `X` to our DBMs, we are able to form the free locale over the set of DBMs. We then use Lean’s support for quotienting to quotient out the unsolvable DBMs as described in Section 5.2.1. The `ClockLocale` shown in Figure 5.4 represents our topological semantics.

```
variable [X : Type] [PartialOrder X]

-- result 1: downsets of poset form a poset instance
downset_of_poset_is_poset : PartialOrder (LowerSet X) where ...

-- result 2: downsets of poset form a frame instance
downset_frame : Frame (LowerSet X) where
  -- downset_frame applied to this type confers the free locale structure
  abbrev
  FreeDBM := LowerSet (DBM n)
```

Figure 5.3: We formalize two results from Locale Theory, namely that the lower closed sets of a partial order and that lower closed sets form a Locale. Note that Mathlib uses the terminology “Frame” instead of Locale.

```

-- we quotient the free frame by the following relation which associates to
-- every collection of DBMs (join of formal opens) its convex hull
inductive cover : (FreeDBM n) → (FreeDBM n) → Prop where
| hull {s : Set (LowerSet (DBM n))} : cover (sSup s) (@lowerClosure (DBM
  n) _ (@convexHull ℕ (DBM n) _ _ _ (sSup s).carrier))

-- the congruence closure over the cover relation instance DBMSetoid :
Setoid
(FreeDBM n) where r := Relation.EqvGen (cover n) iseqv :=
  Relation.EqvGen.is_equivalence (cover n)

-- finally we obtain the Clock locale by taking the quotient of the free
-- frame
-- by the congruence closure
def ClockLocale := Quotient (DBMSetoid n)

```

Figure 5.4: The construction of the Clock Locale takes advantage of Lean’s support for quotient types as well as congruence closures. We finalize the formalization with an axiom

There is one remaining central issue, which is that we must make a connection between this type and that of DBMs in particular. We have left out implementing explicit machinery for keeping track of solution sets of DBMs, which means that we lose access to the piece of data that *is* the convex structure holding the set of valuations. Connecting the two is straightforward to implement and is part of ongoing work to make the library more robust. For now, we keep the formalization of the Clock Locale and the Refinement library separate.

5.4.2 Zones and DBMs

The starting reference for zones and difference bound matrices (DBMs) is the work of Dill in [51]. This paper builds a language of inequality constraints, terms of the form $(\{\leq, <\}, ez)$ where ez is an element of the extended integers (the integers with positive and negative infinity attached). He then encodes systems of these inequality constraints as square matrices, the DBMs. The work was expanded upon in subsequent papers and the work in [5] as well as the UDBM library [56] are accessible treatises on how to work

with DBMS. Dill’s original work has not been, to our knowledge, formalized in a theorem prover, and the UDBM library is a C library which focuses more on implementation. We are aware of one existing formalization of DBMs, the work of Wimmer and Lammich in [57] which was done in Isabelle/HOL, however it was not consulted in PLATA.

We begin by building the extended integers as well as the inequality encoding. We do this incrementally, first building the product of the integers with the booleans to create pairs of numbers with either $\leq$ (false) or $<$ (true). We then attach a bottom element and finally a top element. This avoids the creation of nonexistent elements, as would occur with just using Lean’s built-in extended integers. For each structure, from `Z0` to `Bound` we prove several structural theorems, including instances of orders as well as homomorphism imbeddings from each lower structure to the one above it, e.g. from `Z0Bot` to `Bound`. This culminates in a proof of the fact that this structure, the `Bound` forms a min-plus semiring, often called a “tropical semiring”. That is, `Bound` have a minimum operation as well as an addition operation, which obey the laws of a semiring [58], shown in Figure 5.5.

```
abbrev ops := Bool

def Z0 := ℤ × ops
  deriving DecidableEq

abbrev Z0Bot := WithBot Z0

abbrev Bound := WithTop Z0Bot

instance : LinearOrderedAddCommMonoidWithTop Bound where ...

#synth SemilatticeSup (Tropical Bound)

#synth CommSemiring (Tropical Bound)
```

Figure 5.5: The basic definitions required to build `Bounds` as well as the proof that `Bounds` form a tropical semiring.

On top of the `Bound` structure we can build DBMs as square matrices. DBMS are

matrices of dimension $(1 + |\text{Clk}|) \times (1 + |\text{Clk}|)$ where “Clk” is the set of clocks. The added dimension refers to a reference clock, or zero clock, which constitutes the first row and column of the matrix, useful for expressing singular time constraints; e.g., $x \leq 4$ would be written as $x - 0 \leq 4$. The cells in a DBM are a pair of a comparison operator, `ops`, and an `Bound`. If `(1e, (some(some 4)))` is the bound of the cell at position (i, j) , then we read this as $\text{clk}_i - \text{clk}_j \leq 4$. Operations on DBMs are the pointwise extension from Bounds, namely intersection and sum, and they are partially ordered as well. Additionally, DBMs form a meet semilattice. In fact, DBMs also form an additive commutative monoid. There are several operations we can perform on DBMs, which feature prominently in the model checking community. We implement them here as “clock transformers”. The main operations are `elapse`, `reset`, and `intersect_with_guard`. Figure 5.6 shows the pertinent definitions.

The remaining operations are implemented similarly, with algorithms taken from [5] and [56]. As discussed in Section 5.2.1, we currently do not automatically ensure that DBMs are solvable (i.e., describe convex sets), leaving it up to the user to ensure convexity, but we are working on implementing these checks.

5.4.3 Refinement Calculus

The Refinement Calculus was independently developed by Back and von Wright in [21] as well as Morgan in [22]. The former development is based more on the mathematics of lattices while the latter is rooted more in the practice of programming, concretely implementing code from specifications. The work of Alpuim and Swierstra [59] is an implementation in Rocq of the Morgan-style Refinement Calculus. We largely follow their presentation, making a few additions based on useful operators from the Back-von-Wright-style. The main definitions are that of the predicate transformer `PT` and

```

(ClockSet : ℕ)

abbrev DBM (n : ℕ) := Matrix (Fin (1 + n)) (Fin (1 + n)) (Tropical Bound)

instance DBM_zero : Zero (DBM n) := Matrix.zero

instance DBM_one : One (DBM n) := Matrix.one

instance DBM_SemilatticeSup {n : ℕ} : SemilatticeSup (DBM n) := ...

lemma DBM_add_self (d : DBM n) : d + d = d := ...

def intersect_with_guard (z : (Fin (1 + ClockSet) × Fin (1 + ClockSet)) ×
  Bound)
  (d : DBM ClockSet) : DBM ClockSet := ...

def elapse_all {n : ℕ} (d : DBM n) : DBM n := ...

def reset {n : ℕ} (rc : Fin (1 + n)) (val : ℤ) (d : DBM n) : DBM n := ...

```

Figure 5.6: DBMs are square matrices of bounds of dimension 1 more than the number of clocks. They inherit sum and intersection from the Bounds, as well as a partial order. On top of the DBM structure we define the usual operations, such as resetting clocks, letting time elapse, and intersecting constraints.

refinement, `Refine`, as shown in Figure 5.7.

The refinement calculus presented in [59] is closer in flavor [22] and thus omits operations found in Back and von Wright, such as the “assumption” and “assertion” operators, as well as operations found in Dijkstra’s Guarded Command Language (GCL), such as the “fat bar” nondeterministic conditional. We take pains to close this gap, and in Figure 5.10 show an example defining the conditional operator of the GCL in the style of Morgan and Alpuim.

5.4.4 Discussion

We have built a framework for constructing spaces in which one can reason about timed, concurrent processes by thinking topologically about the denotational semantics

```

structure PT : Type where
  pre : Set S
  post :  $\forall (s : S), \text{pre } s \rightarrow A \rightarrow \text{Set } S$ 

def PT_meet (pt1 pt2 : PT S A) : PT S A := by
  apply PT.mk (pt1.pre  $\cap$  pt2.pre)
  · intro s k a
    have <kl,kr> := k
    exact (pt1.post s kl a)  $\cap$  (pt2.post s kr a)

instance PT_min : Min (PT S A) where
  min := PT_meet S A

inductive Refine (pt1 pt2 : PT S A) : Prop where
| Refinement :
   $\forall (d : \text{pt1.pre} \subseteq \text{pt2.pre}),$ 
   $(\forall (s : S) (x : \text{pt1.pre } s) (v : A),$ 
     $(\text{pt2.post } s (d \ x)) \ v \subseteq (\text{pt1.post } s \ x \ v)) \rightarrow$ 
    Refine pt1 pt2

def refineTransPT (pt2 pt1 pt3) :
  Refine S A pt1 pt2  $\rightarrow$  Refine S A pt2 pt3  $\rightarrow$  Refine S A pt1 pt3 := by
  intro h1 h2
  cases h1 with
  | Refinement pre12 pt21 =>
    cases h2 with
    | Refinement pre23 pt32 =>
      apply Refinement
      · intro s' x' (v' : A) _ k
        have t2 := pt21 s' x' v'
        have t3 := pt32 s' (pre12 x') v'
        exact (t2 (t3 k))

```

Figure 5.7: We port the basic definitions of [59], expanding on the algebraic structure slightly by, e.g., providing lattice operations such `meet`.

of timed, concurrent systems . We did this this by representing sets of temporal traces as convex regions and representing convex regions as special matrices, DBMs. We then situated this in the Refinement Calculus (RC). By defining spaces according to pre and post conditions, where the conditions are the timing constraints, the RC becomes a readily accessible tool for programming reactive systems in which the correctness properties are built-in at the outset, as opposed to verified after the fact. This work is ongoing while we

```

def assumptionA (g : S → Bool) : PT S A := by
  let asmPre :=
    λ s => if (g s) then True else False
  apply PT.mk asmPre
  · intro s k a ; exact univ

def CondPT (c : S → Bool) (pt1 pt2 : PT S A) : PT S A := by
  have pos_cond := assumptionA S A c
  have neg_cond := assumptionA S A (not ∘ c)
  have left_arm := SeqPT S A A pt1 pos_cond
  have right_arm := SeqPT S A A pt2 neg_cond
  exact left_arm ⊓ right_arm

```

Figure 5.8: We unify the two presentations of the Refinement Calculus, due to [21] and [22] giving the user access to a wider set of specification and implementation tools.

```

inductive Refine' (clk : ℕ) (pt1 pt2 : PT' A clk) : Prop where
| Refinement' :
  ∀ (d : pt1.pre ⊆ pt2.pre),
  (∀ (s : DBM clk) (x : s ⊆ pt1.pre) (v : A),
    (pt2.post s (DBM_subset_trans x d)) v ⊆ (pt1.post s x v)) →
    Refine' clk pt1 pt2

```

Figure 5.9: The Refinement relation adapted to DBMs intuitively reflects that a precondition refines another if we can increase (or maintain) the number of available actions and decrease (or maintain) the number of known outcomes.

come up with more and larger examples on which to work; however, it represents a novel approach which we believe suggests interesting contributions on both the theoretical and practical side of reactive systems.

```

-- initial specification as a pred trans
abbrev resetDBMSpec : PT' Unit 2 := by
  let frobPre : DBM 2 := -- some DBM
  let frobPost (a:DBM 2) (pre:relationp a frobPre) (u:Unit) : DBM 2 := -- " "
  exact PT'.mk frobPre frobPost

-- lifting DBM functions to pred trans
def resetDBM : PT' Unit 2 := by
  let resetPre : DBM 2 :=
    univ_dbm
  let resetPost:(a:DBM 2) → relationp a resetPre → Unit → DBM 2 := by
    intro a pre _
    exact (@reset 2 1 0 a) -- a function over DBMs
  exact PT'.mk resetPre resetPost

-- incrementally prove a refinement over two pred trans
def reset_spec_prog :
  ∃ (c : PT' Unit 2), Refine' Unit 2 resetDBMSpec c := by
  ...

```

Figure 5.10: We first establish a specification based on pre- and postconditions. We give relevant functions also in terms of pre- and postconditions. This allows us to the type of a refinement.

Chapter 6

Ongoing and Future Work

PLATA was designed to make specifying systems behaviorally much simpler and currently this finds usecases most naturally in constraining timing constraints, namely, building spaces in which actions are “well-timed” in some sense. PLATA nevertheless has a broad capacity to describe concurrency, inherited from operators of the Refinement Calculus, such as the nondeterministic conditional. Concurrency in this manner is exemplified by complicated interleavings of processes, ensuring specific actions or events occur in the correct order. In an attempt to fortify PLATA in this area, we have begun a project to study a selection of concurrent algorithms, specifically focusing on communication protocols. To do this, we formalize this study in the language of Process Algebra [60]. The goal is then to port this formalization back into the topological framework established in PLATA. While this is ongoing work, we have made some interesting results on which we can report below.

6.1 Process Algebra and Communication Protocols

Process algebras are specifications of concurrent and parallel systems using sets of equations. The central idea is to treat processes as variables $P, Q, R, \dots$ and operators, such as choice $P + Q$, sequential composition $P \cdot Q$ and parallel composition $P || Q$, as functions over these variables. An expression $R = P + Q$ would stand for “process R can behave as P and terminate or Q and terminate”. As in standard algebra, these operators, to be meaningful, must be constrained by their own axioms. For instance, we might want axioms stating that a choice of processes is commutative, associative, and distributes on the left over sequential composition. This is expressed in the set of equations shown in Figure 6.1.

$$\begin{aligned}(P + Q) &= (Q + P) \\ P + (Q + R) &= (P + Q) + R \\ (P + Q) \cdot R &= (P \cdot R) + (Q \cdot R)\end{aligned}$$

Figure 6.1: Some examples of process algebra equations. The plus sign is used to indicate a choice of either process on the left or right. The center dot indicates a sequence of processes.

Historically, the Algebra of Communicating Processes (ACP) and the Calculus of Communicating Systems (CCS) are important process algebras [60]. We will focus on a modern incarnation of ACP, namely mCRL2 [54]. This language boasts a large suite of automated tooling, from a core language, to a model checker, to visualization tools. Its version of process algebra consists of a large class of operators, including those allowing for channel-based communication and recursive definitions.

mCRL2

The process algebra mCRL2 implements ACP, henceforth just “process algebra”, with additions for data and timing. That is, it additionally includes axioms for dealing with finite (and in some cases, inductively defined) data domains, as well as time. In practice, however, the support for timing is limited, as all time-sensitive equations must be discretized, and to some degree, this task is left to the user as opposed to the tool. To use mCRL2, the user writes `.mcr12` files which contain sets of process algebra equations. These files can then be simulated allowing the user to step through the available actions in the system. Notably however, mCRL2 does not allow the user to reason about the process equations axiomatically. All of the tools in the mCRL2 tool suite convert the set of equations to an internal normal form representation called a “linear process specification” (LPS).

The tools in the suite are accessible as binaries which perform actions on this internal representation. For instance, *mcr122lps* is the tool that converts the written process algebra equations to the internal normal form and *lps2lts* synthesizes from the normal form a labeled transition system (LTS). The main operations mCRL2 supports are model checking-based. mCRL2 supports a model checker for modal μ -calculus, a very expressive modal logic with fixed points (also extended with operations for data and timing). Model checking an LPS or LTS against a formula ϕ is done through a parameterized boolean equation system (PBES). The combination of the formula and the model (LPS or LTS) are converted to a set of boolean equations which are then solved using automatic means, such as alternating automata, or parity games, etc. This choice of solution methodology, while very efficient for model checking queries, makes equivalence checking difficult and non-obvious.

6.1.1 Communication Protocols

Communication protocols describe algorithms in which multiple actors (process, servers, etc.) can exchange information. Classic examples include the alternating bit protocol [54], bounded retransmission [26], and fair leader election [27]. The process algebra community has devoted a lot of effort to specify and prove these protocols formally correct [28, 29]. This work has been predominantly manual, in the sense that the equations have been developed by hand and proven in pen and paper derivations. As an example of a very simple communication protocol, we look at the 2-place buffer, visualized in Figure 6.3. A 2-place buffer is intuitively a queue with capacity for 2 elements. Instead of allocating memory in the style of an array, for instance, the 2-place buffer is a system of parallel processes which perform the actions of pushing and popping elements through from source to target. This design can be realized in mCRL2 and is shown in Figure 6.4. The mCRL2 toolsuite then compiles this set of equations into an automata so that it can be model checked, pictured in Figure 6.5. This synthesis process is detailed in [61]. We note that this is an interesting departure from process algebra. The theory of solving process algebra equations is avoided and replaced with traditional model checking techniques.

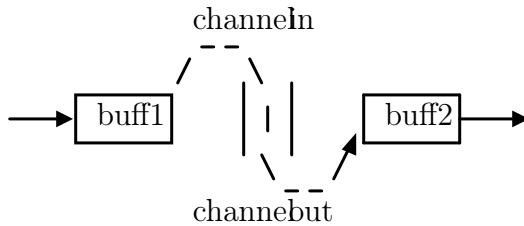


Figure 6.2: This depicts the 2-Place Buffer protocol.



Figure 6.3: This depicts a queue of depth 2.

The code in Figure 6.4 can be described as follows. B1 and B2 are the process equations which are put in parallel as $B1 \parallel B2$. `act` sets up the signals needed to communicate

```

sort
    D      = struct d1 | d2;
act
    r , s : D;
    ri , si , i : D;
proc
    B1 = sum d : D . r(d) . si(d) . B1;
    B2 = sum d : D . ri(d) . s(d) . B2;
init
    allow({r,s},
        comm({si|ri->i},
            B1 || B2
        )
    );

```

Figure 6.4: This is mCRL2 code as the user would write to implement the 2-place buffer. **B1** and **B2** are the process equations which are put in parallel as **B1 || B2**. **act** sets up the signals needed to communicate and the **allow** and **comm** syntax realizes this in the parallel system.

and the **allow** and **comm** syntax realizes this in the parallel system. **sort** establishes the type of data being pushed through the processes. In total, we think of **B1** waiting for a signal of type **D** which could contain a **d1** or **d2**, receiving it on the **r** channel (or *as* an **r** signal), then following the sequence operator, and sending it on the **si** channel, finally, recursively returning to its “initial state”. In parallel, the **B2** process awaits a signal on the **ri** channel which it will forward on the **s** channel before similarly continuing as itself. The channels are plumbed together via the **allow** and **comm** keywords.

The labeled transition depicted in Figure 6.5 is interpreted as a simple, deterministic finite state automaton. It shows all of the possible sequences of messages that the 2-place

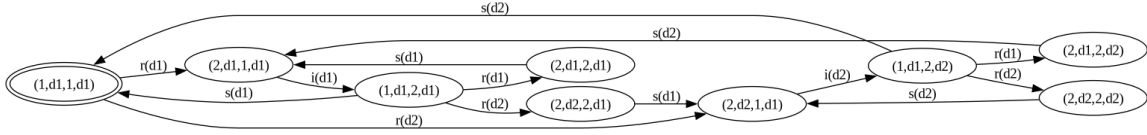


Figure 6.5: The process algebra equations can be compiled into a labeled transition system as described in [61]. While this avoids using the theory of process algebra, it does provide the benefit of sophisticated model checking.

buffer can realize. The edges indicate sending and receiving D struct values. mCRL2 model checks this automaton according to queries written in the modal μ -calculus .

6.2 Automating Process Algebra

Model checking communication protocols represents a high degree of automated verification. It is worth asking however, how far we can get with process algebra alone. We observe that the two technologies, model checking finite automata and solving process algebra equations, offer distinctly different advantages and disadvantages. In fact the situation resembles very much that of PyLSE-TA and Citrus. While Citrus did not replace PyLSE-TA by any means, there were clear advantages to curtailing state space explosion by recasting the verification problems as algebraic equations and then solving them (in the case of Citrus, via coinduction). We encode a small subset of process algebra, as described in [60] in Lean using quotient types. We then leverage Lean’s metaprogramming facilities and solver engines, such as `aesop` and `grind` [62, 63] to automatically deduce the correctness of process algebra equations. Beyond the work of bij ’t Vuur and Coates [64, 65], we do not know of any attempts at formalizing process algebra. We show three code snippets below to demonstrate our encoding.

In Figure 6.6 we encode a subset of process algebra as a simple inductive type, feature a deadlock (or non-termination) symbol, as well as sequential and branching operators.

We then encode the axiom relation as an inductive type as well, indexed over the language `alg` as a binary relation. We then take advantage of Lean’s language features to turn the base language and its axiom relation into a Setoid object and then a Quotient type. Encoding process algebra as a Quotient type has proven useful for two reasons. It is the easiest way to flexibly add new axioms, and it allows us to use Lean’s built-in equality, as opposed to a specific Setoid relation.

```

inductive alg : Type where
| dlk      : alg -- deadlock
| seq      : ∀ (_ _ : alg), alg
| br       : ∀ (_ _ : alg), alg

inductive q : alg → alg → Prop where
-- basic axioms
| br_idem   : ∀ (a : alg), q (a + a) a
| br_comm   : ∀ (a b : alg), q (a + b) (b + a)
| br_assoc  : ∀ (a b c : alg), q ((a + b) + c) (a + (b + c))
| seq_assoc : ∀ (a b c : alg), q ((a · b) · c) (a · (b · c))
| distrib_r : ∀ (a b c : alg), q ((a + b) · c) ((a · c) + (b · c))
-- axioms for deadlock
| br_deadlock : ∀ (a : alg), q (a + δ) a
| seq_deadlock : ∀ (a : alg), q (δ · a) δ
-- congruence axioms
| br_congr_l  : ∀ (a b c : alg), (q a b) → q (a + c) (b + c)
| br_congr_r  : ∀ (a b c : alg), (q b c) → q (a + b) (a + c)
| seq_congr_l : ∀ (a b c : alg), (q a b) → q (a · c) (b · c)
| seq_congr_r : ∀ (a b c : alg), (q b c) → q (a · b) (a · c)

```

Figure 6.6: We can encode process algebra as a simply inductive type and the axioms as a binary relation over the language. We map the `seq` and `br` operators to new symbols. Note that we need to add congruence axioms additionally which are typically left unspecified in process algebra presentations.

In Figure 6.7 we turn the binary axiom relation into a quotient type using Lean’s support for setoids. All operations defined on the base algebra, the `alg` type, must be lifted to the Quotient type, or rather, proved to be well-formed. It is here specifically that we need the congruence laws.

Finally in Figure 6.8, we show that we can prove equations in the style of process

```

instance Alg : Setoid alg where

def Action := Quotient Alg

def alg_add (x y : alg) : Action :=
  Quotient.mk Alg (x + y)

@[grind =]
instance Action_Add : Add Action where
  add A :=
    A.lift₂ alg_add <| by
      intro a1 b1 a2 b2 h1 h2
      apply Quotient.sound
      have t1 := br_congr_l a1 a2 b1 h1
      have t2 := br_congr_r a2 b1 b2 h2
      exact q_trans _ _ _ t1 t2

```

Figure 6.7: We can lift the syntax from the original process algebra type into the quotient type. This turns all operations defined in the process algebra, e.g. `seq` into well-formed operations that work over corresponding quotient objects.

algebra. We can do so manually, as in the proof for `Action.deadlock_mul`, in which we show that deadlocking followed by an action is the same as deadlocking. The proof in `Action.example_red` shows a simple reduction in which an equality may be used to deduce that only the process `A` may be performed. Both proofs can be solved manually, using tactics, such as `simp` and `rw` which respectively simplify definitions and rewrite by known proofs. However, as the last two `examples` show, the proofs can be entirely automated using Lean’s built in `grind` solver, which performs internally a series of rewrites, deductions, proof strategies, etc.

The end result, while still preliminary, is that by encoding process algebra in a sophisticated theorem prover, we are in a position to not only formalize many hand-written proofs of communication protocols, but also do so with powerful automated solver facilities. It is an open question whether these proofs can be entirely automated or if some amount of manual direction is still required; however, the above approach allows for both to seamlessly interoperate.

```

--  $\Delta$  is the lifted dlk, or deadlock, symbol

@[grind =]
lemma Action.deadlock_mul (A : Action) : ( $\Delta$  A) * A = ( $\Delta$  A) := by
  cases A using Quotient.ind
  apply Quotient.sound
  apply seq_deadlock

-- (A = B)  $\vdash$  ((A + A) + B)
-- -----
-- A
lemma Action.example_red (A B : Action) (h : A = B) : ((A + A) + B) = A := by
  have AeqA := Action.add_idem A
  have Acongr := (@Action.add_congr_1 (A + A) A B AeqA)
  rw[Acongr ,  $\leftarrow$ h, AeqA]

-- the same proof but `grind` solves automatically
example (A B : Action) (h : A = B) : ((A + A) + B) = A := by
  grind

example (A : Action) : ( $\Delta$  A) * (A + ( $\Delta$  A)) = ( $\Delta$  A) := by
  grind

```

Figure 6.8: With the above encoding of process algebra we can manually as well as automatically solve process algebra equations. This opens the door to formalizing a large amount of concurrent algorithms just as well as design and verify new ones.

6.3 Discussion

As we mentioned above in Section 6.1.1, many communication protocols have been manually verified in process algebra. This represents a substantial step in obtaining confidence in our ability to analyze these concurrent algorithms. Nevertheless, the proofs are still unformalized, or not formally checked themselves. Modern theorem provers are performant enough to remedy this, and in most cases, expand on the original framework through automation and metaprogramming facilities. Process algebra has not recently received a lion's share of attention in the programming language or formal verification community, but has a great deal to offer in this space.

With the exception of bij 't Vuur and Coates [64, 65], we know of no other antecedents

in the formalization of process algebra or the verification of process algebra proofs of concurrent algorithms. We further break with the work of <name> by avoiding the automata-theoretic semantics of process algebra entirely, focusing solely on the equational presentation. We do this by leveraging Lean’s quotient types, currently a relatively unique feature of Lean among other theorem prover languages. We further adapt Lean’s metaprogramming and solver facilities. This allows process algebra proofs to be written in a completely manual fashion, as in the original verification papers, but also in an automated fashion, at times bypassing manual proofs entirely.

Our formalization additionally casts light on existing verification methods for concurrent algorithms. The mCRL2 tool provides a high degree of model checking tooling, however, this requires compiling the process algebra specifications into automata. This suffers from all of the problems mentioned in previously, for instance in Section 3.3 and Section 4.1. Much in the manner that Citrus was able to simplify verification problems over automata into algebraic equation solving, and thus obtain a large computational benefit, we conjecture that many verification problems over concurrent algorithms benefit from the process algebra approach.

Finally, the study, while still ongoing, serves as a foundation on which to expand PLATA. Specifically, the framework in which many of the protocols were designed and proved in process algebra mimics the style of PLATA, by establishing pre and post conditions, and incrementally refining the equations. Few existing communication protocols are formalized with explicit notions of timing, despite the fact that timing is, in any implementation setting, a core feature difficult to abstract away. We see this as an important contribution to the research in formal verification of concurrent algorithms.

Chapter 7

Conclusion

In this work we have described three novel languages for describing reactive systems, with a focus on timing and concurrency. The first, PyLSE-TA, is a specification and verification language specifically for superconducting electronics. It is based on translating specifications of circuits into timed automata and schematizing correctness properties in the logic TCTL. We then presented Citrus, that extends PyLSE-TA into a fully-fledged programming language by providing a set of functional building blocks, as well as expanding the proof facilities, by formalizing equivalence between circuits. Finally, we presented Plata, a calculus for modeling and equationally reasoning generally about stateful, concurrent, nondeterministic, timing-sensitive specifications. PLATA is complementary to the standard approach for dealing with such systems via Timed Automata and model checking. Plata provides a novel semantics as well as several verification tools for the designer of reactive systems. We showed that Citrus and PLATA, by using techniques rooted in type systems and denotational semantics, offer several advantages over post-hoc verification methods. Citrus avoided a large amount of state space explosion through compositional design while PLATA affords building systems in a “correct by construction” manner. A major result of these investigations is that reactive systems, while histori-

cally represented by model checking and post-hoc verification, can be adapted to a much wider set of analysis and verification techniques, which in many cases, provides not just a novel perspective, but concrete improvements. In future work, we would like to apply the above languages to larger problems in distributed and concurrent systems, specifically focusing on communication protocols and consensus algorithms, including those at the hardware-software interface.

Bibliography

- [1] M. Christensen, G. Tzimpragos, H. Kringen, J. Volk, T. Sherwood, and B. Hardekopf, “Pylse: a pulse-transfer level language for superconductor electronics,” in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 671–686. [Online]. Available: <https://doi.org/10.1145/3519939.3523438>
- [2] R. Nigam, P. H. Azevedo de Amorim, and A. Sampson, “Modular hardware design with timeline types,” *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, Jun. 2023. [Online]. Available: <https://doi.org/10.1145/3591234>
- [3] G. Tzimpragos, A. Madhavan, D. Vasudevan, D. Strukov, and T. Sherwood, “Boosted race trees for low energy classification,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 215–228. [Online]. Available: <https://doi.org/10.1145/3297858.3304036>
- [4] R. Gretsches, M. Beyeler, J. Lau, and T. Sherwood, “Single spike artificial neural networks,” in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA ’25. New York, NY, USA: Association for Computing Machinery, 2025, p. 916–929. [Online]. Available: <https://doi.org/10.1145/3695053.3731027>
- [5] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi, “Uppaal — a tool suite for automatic verification of real-time systems,” in *Hybrid Systems III*, R. Alur, T. A. Henzinger, and E. D. Sontag, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 232–243.
- [6] P. C. Kocher, “Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems,” in *Annual International Cryptology Conference*, 1996. [Online]. Available: <https://api.semanticscholar.org/CorpusID:268055936>
- [7] T. Brennan, N. Rosner, and T. Bultan, “Jit leaks: Inducing timing side channels through just-in-time compilation,” in *Proceedings - 2020 IEEE Symposium on Se-*

curity and Privacy, SP 2020, ser. Proceedings - IEEE Symposium on Security and Privacy, May 2020, pp. 1207–1222, publisher Copyright: © 2020 IEEE.; 41st IEEE Symposium on Security and Privacy, SP 2020 ; Conference date: 18-05-2020 Through 21-05-2020.

- [8] R. Alur and D. L. Dill, “A theory of timed automata,” *Theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994.
- [9] E. M. Clarke, E. A. Emerson, and A. P. Sistla, “Automatic verification of finite-state concurrent systems using temporal logic specifications,” *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, p. 244–263, Apr. 1986. [Online]. Available: <https://doi.org/10.1145/5397.5399>
- [10] A. Pnueli, “The temporal logic of programs,” in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, 1977, pp. 46–57.
- [11] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.
- [12] G. Behrmann, A. David, and K. G. Larsen, “A tutorial on uppaal 4.0,” Aalborg University, Denmark, Department of Computer Science, Tech. Rep., 2008.
- [13] J. A. Delport, K. Jackman, P. I. Roux, and C. J. Fourie, “Josim—superconductor spice simulator,” *IEEE Transactions on Applied Superconductivity*, vol. 29, no. 5, pp. 1–5, 2019.
- [14] J. Hughes, “Programming with Arrows,” in *Advanced Functional Programming*, V. Vene and T. Uustalu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, vol. 3622, pp. 73–129, series Title: Lecture Notes in Computer Science. [Online]. Available: https://link.springer.com/10.1007/11546382_2
- [15] C. Elliott and P. Hudak, “Functional reactive animation,” *SIGPLAN Not.*, vol. 32, no. 8, p. 263–273, aug 1997. [Online]. Available: <https://doi.org/10.1145/258949.258973>
- [16] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson, *Arrows, Robots, and Functional Reactive Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 159–187. [Online]. Available: https://doi.org/10.1007/978-3-540-44833-4_6
- [17] A. Bove, P. Dybjer, and U. Norell, “A brief overview of agda – a functional language with dependent types,” in *Theorem Proving in Higher Order Logics*, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 73–78.
- [18] H. Liu, E. Cheng, and P. Hudak, “Causal commutative arrows and their optimization,” *SIGPLAN Not.*, vol. 44, no. 9, p. 35–46, aug 2009. [Online]. Available: <https://doi.org/10.1145/1631687.1596559>

- [19] L. Aceto, A. Ingolfsdottir, K. G. Larsen, and J. Srba, *Reactive Systems: Modelling, Specification and Verification*. Cambridge: Cambridge University Press, 2007. [Online]. Available: <http://ebooks.cambridge.org/ref/id/CBO9780511814105>
- [20] L. d. Moura and S. Ullrich, “The lean 4 theorem prover and programming language,” in *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 625–635. [Online]. Available: https://doi.org/10.1007/978-3-030-79876-5_37
- [21] R.-J. Back and J. Wright, “Refinement calculus: A systematic introduction,” *Texts in Computer Science*, 1998.
- [22] C. Morgan, *Programming from specifications*. USA: Prentice-Hall, Inc., 1990.
- [23] P. T. Johnstone, B. Banaschewski, M. Fourman, M. Hyland, and J. Isbell, “The point of pointless topology,” *Bulletin of the American Mathematical Society*, vol. 8, pp. 41–53, 1983. [Online]. Available: <https://api.semanticscholar.org/CorpusID:15371069>
- [24] S. Vickers, *Topology via logic*. USA: Cambridge University Press, 1989.
- [25] M. Bonsangue, “Topological dualities in semantics,” PhD-Thesis - Research and graduation internal, Vrije Universiteit Amsterdam, 1996.
- [26] J. Groote and J. Pol, “A bounded retransmission protocol for large data packets: A case study in computer checked algebraic verification,” 12 1995.
- [27] L. Åke Fredlund, J. F. Groote, and H. Korver, “Formal verification of a leader election protocol in process algebra,” *Theoretical Computer Science*, vol. 177, no. 2, pp. 459–486, 1997. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0304397596002563>
- [28] C. P. J. Koymans and J. Mulder, “A modular approach to protocol verification using process algebra,” *Logic group preprint series*, vol. 6, 1986. [Online]. Available: <https://api.semanticscholar.org/CorpusID:62336266>
- [29] *Applications of Process Algebra*, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.
- [30] H. Hayakawa, N. Yoshikawa, S. Yorozu, and A. Fujimaki, “Superconducting digital electronics,” *Proceedings of the IEEE*, vol. 92, no. 10, pp. 1549–1563, oct 2004. [Online]. Available: <http://ieeexplore.ieee.org/document/1335548/>
- [31] P. Bouyer, P. Gastin, F. Herbreteau, O. Sankur, and B. Srivathsan, “Zone-based verification of timed automata: Extrapolations, simulations and what next?” in *Formal Modeling and Analysis of Timed Systems: 20th*

- International Conference, FORMATS 2022, Warsaw, Poland, September 13–15, 2022, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2022, p. 16–42. [Online]. Available: https://doi.org/10.1007/978-3-031-15839-1_2
- [32] J. Hughes, “Generalising monads to arrows,” *Science of Computer Programming*, vol. 37, no. 1, pp. 67–111, 2000. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167642399000234>
- [33] R. Paterson, “A new notation for arrows,” in *International Conference on Functional Programming*. ACM Press, Sep. 2001, pp. 229–240. [Online]. Available: <http://www.soi.city.ac.uk/~ross/papers/notation.html>
- [34] C. Paulin-Mohring, *Introduction to the Coq Proof-Assistant for Practical Software Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 45–95. [Online]. Available: https://doi.org/10.1007/978-3-642-35746-6_3
- [35] Y. Bertot and E. Komendantskaya, “Inductive and coinductive components of corecursive functions in coq,” *Electronic Notes in Theoretical Computer Science*, vol. 203, no. 5, pp. 25–47, 2008, proceedings of the Ninth Workshop on Coalgebraic Methods in Computer Science (CMCS 2008). [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1571066108003320>
- [36] A. Abel, “MiniAgda: Integrating sized and dependent types,” *Electronic Proceedings in Theoretical Computer Science*, vol. 43, pp. 14–28, dec 2010. [Online]. Available: <https://doi.org/10.4204/2Feptcs.43.2>
- [37] J. R. Munkres, *Topology*, 2nd ed. Prentice-Hall Inc., 2000.
- [38] M. O’Connor, “A logical interpretation of some bits of topology.” [Online]. Available: <https://xorhammer.com/2011/07/09/a-logical-interpretation-of-some-bits-of-topology/>
- [39] M. R. Clarkson and F. B. Schneider, “Hyperproperties,” in *2008 21st IEEE Computer Security Foundations Symposium*, 2008, pp. 51–65.
- [40] B. Alpern and F. B. Schneider, “Defining liveness,” *Information Processing Letters*, vol. 21, no. 4, pp. 181–185, 1985. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/0020019085900560>
- [41] E. W. Dijkstra, “Guarded commands, nondeterminacy and formal derivation of programs,” *Commun. ACM*, vol. 18, no. 8, p. 453–457, Aug. 1975. [Online]. Available: <https://doi.org/10.1145/360933.360975>
- [42] J. Rutten, *The Method of Coalgebra: exercises in coinduction*, Feb. 2019. [Online]. Available: <https://ir.cwi.nl/pub/28550/>

- [43] B. Jacobs, *Introduction to Coalgebra: Towards Mathematics of States and Observation*, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2016.
- [44] M. Hasegawa, *Recursion from Cyclic Sharing*. London: Springer London, 1999, pp. 83–101. [Online]. Available: https://doi.org/10.1007/978-1-4471-0865-8_7
- [45] D. Sangiorgi, *Introduction to Bisimulation and Coinduction*. Cambridge: Cambridge University Press, 2011. [Online]. Available: <http://ebooks.cambridge.org/ref/id/CBO9780511777110>
- [46] A. Abel, “Equational Reasoning about Formal Languages in Coalgebraic Style,” p. 38, 2016.
- [47] R. Cartwright, R. Parsons, and M. A. AbdelGawad, “Domain theory: An introduction,” *CoRR*, vol. abs/1605.05858, 2016. [Online]. Available: <http://arxiv.org/abs/1605.05858>
- [48] P. Di Gianantonio, “Real number computability and domain theory,” *Information and Computation*, vol. 127, no. 1, pp. 11–25, 1996. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0890540196900462>
- [49] B. Srivathsan, “Examples of zone graphs,” course notes, Chennai Mathematical Institute, India, 2013, course notes from webpage, Automata for Real-time Systems: <https://www.cmi.ac.in/~sri/Courses/TIME2013/>. [Online]. Available: <https://www.cmi.ac.in/~sri/Courses/TIME2013/Slides/zone-graph-examples.pdf>
- [50] J. Avigad, K. Buzzard, R. Y. Lewis, and P. Massot, “Mathematics in lean, chapter 6: Topology.” [Online]. Available: <https://www.imo.universite-paris-saclay.fr/~patrick.massot/mil/index.html>
- [51] D. L. Dill, “Timing assumptions and verification of finite-state concurrent systems,” in *Automatic Verification Methods for Finite State Systems*, J. Sifakis, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 197–212.
- [52] A. Tosun, “Formal topology in univalent foundations,” 2020, unpublished master’s thesis. [Online]. Available: <https://odr.chalmers.se/server/api/core/bitstreams/84989823-7398-4eb3-8804-1f88822d1fd2/content>
- [53] G. Manuell, “Uniform locales and their constructive aspects,” 2024. [Online]. Available: <https://arxiv.org/abs/2106.00678>
- [54] O. Bunte, J. F. Groote, J. J. A. Keiren, M. Laveaux, T. Neele, E. P. de Vink, W. Wesselink, A. Wijs, and T. A. C. Willemse, “The mcrl2 toolset for analysing concurrent systems,” in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Vojnar and L. Zhang, Eds. Cham: Springer International Publishing, 2019, pp. 21–39.

- [55] —, “The mcrl2 toolset for analysing concurrent systems,” in *Tools and Algorithms for the Construction and Analysis of Systems*, T. Vojnar and L. Zhang, Eds. Cham: Springer International Publishing, 2019, pp. 21–39.
- [56] M. Mikučionis, R. Rendal, K. Y. Jørgensen, and T. Neustrup, “Uppaal dbm library,” <https://github.com/UPPAALModelChecker/UDBM?tab=readme-ov-file>, 2025.
- [57] S. Wimmer and P. Lammich, “Difference bound matrices,” *Archive of Formal Proofs*, August 2024, https://isa-afp.org/entries/Difference_Bound_Matrices.html, Formal proof development.
- [58] J.-E. Pin, *Tropical semirings*, ser. Publications of the Newton Institute. Cambridge University Press, 1998, p. 50–69.
- [59] J. Alpuim and W. Swierstra, “Embedding the refinement calculus in coq,” *Science of Computer Programming*, vol. 164, pp. 37–48, 2018, special issue of selected papers from FLOPS 2016. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167642317300710>
- [60] R. D. Nicola, “A gentle introduction to process algebras.” [Online]. Available: <https://www.pst.ifi.lmu.de/Lehre/fruhere-semester/sose-2013/formale-spezifikation-und-verifikation/intro-to-pa.pdf>
- [61] J. Groote and M. Mousavi, *Modeling and analysis of communicating systems*. MIT Press, 2014.
- [62] J. Limperg and A. H. From, “Aesop: White-box best-first proof search for lean,” in *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 253–266. [Online]. Available: <https://doi.org/10.1145/3573105.3575671>
- [63] L. de Moura and S. Ullrich, *The Lean Language Reference*, 2024. [Online]. Available: <https://lean-lang.org/doc/reference/latest/>
- [64] W. bij ’t Vuur, “A deep embedding of μcrl in lean,” https://lean-forward.github.io/pubs/vuur_msc_thesis.pdf, 2023.
- [65] J. Coates, “A formalisation of biochemical process languages in lean,” https://project-archive.inf.ed.ac.uk/ug4/20201778/ug4_proj.pdf, 2020.